Shared Memory Abstractions for Heterogeneous Multicore Processors

Scott Schneider

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

Dimitrios S. Nikolopoulos, Chair
Henrique C. M. Andrade
Godmar Back
Kirk Cameron
Calvin J. Ribbens

December 10, 2010
Blacksburg, Virginia

Keywords: EMM, Cell BE, Programming Models, Parallel Programming, Parallel Hardware Architecture

Shared Memory Abstractions for Heterogeneous Multicore Processors

Scott Schneider

We are now seeing diminishing returns from classic single-core processor designs, yet the number of transistors available for a processor is still increasing. Processor architects are therefore experimenting with a variety of multicore processor designs. Heterogeneous multicore processors with Explicitly Managed Memory (EMM) hierarchies are one such experimental design which has the potential for high performance, but at the cost of great programmer effort. EMM processors have cores that are divorced from the normal memory hierarchy, thus the onus is on the programmer to manage locality and parallelism. This dissertation presents the Cellgen source-to-source compiler which moves some of this complexity back into the compiler. Cellgen offers a directive-based programming model with semantics similar to OpenMP for the Cell Broadband Engine, a general-purpose processor with EMM. The compiler implicitly handles locality and parallelism, schedules memory transfers for data parallel regions of code, and provides performance predictions which can be leveraged to make scheduling decisions. We compare this approach to using a software cache, to a different programming model which is task based with explicit data transfers, and to programming the Cell directly using the native SDK. We also present a case study which uses the Cellgen compiler in a comparison across multiple kinds of multicore architectures: heterogeneous, homogeneous and radically data-parallel graphics processors.

This research is or was supported by grants from NSF (CCR-0346867, CCF-0715051, CNS-0521381, CNS-0720750, CNS-0720673), the U.S. Department of Energy (DE-FG02-06ER25751, DE-FG02-05ER25689), IBM, and Virginia Tech (VTF-874197).

Contents

List of Figures

List of Tables

Chapter 1

Introduction: The Potential and Challenges of Heterogeneous Multicore Processors

Processor designs are changing in fundamental ways. In the past, processor architects were able to improve performance by continuing to exploit instruction level parallelism (ILP) and adding to the cache hierarchy. While designers used different techniques to exploit ILP over the years—such as Very Long Instruction Words versus superscalar—the fundamental idea was to squeeze as much performance as possible from a single instruction stream.

These techniques have hit their limit. Long instruction pipelines are needed to keep many instructions in flight at once, using different functional units in parallel. As instruction pipelines grow, they incur several complications. The cost of mispredicted branches increases; it is more difficult to balance the cost of each stage; and they are more prone to stalls from cache misses. Even aggressive out-of-order processors are limited by the amount of ILP that exists in an instruction stream.

Meanwhile, the number of transistors available to processor architects continues to increase, consistent with Moore's Law. Faced with a still increasing transistor budget and old designs based on exploiting ILP that no longer scale, architects are forced to explore new designs. These designs no longer rely solely on ILP, but instead provide hardware amenable to both thread and data level parallelism. These new designs are referred to as *multicore processors* because there are at least two execution cores with distinct execution pipelines, functional units and usually one level of private cache.

Some multicore processors are homogeneous: each of the cores is the same. Such homogeneous multicores typically have an architecture in which multiple sequential cores share a cache. Programming a homogeneous multicore is similar to programming a Symmetric Multiprocessor (SMP), where multiple sequential processors are connected to a single shared

main memory. Message passing paradigms are possible on such multicores, but they are perhaps most naturally programmed with shared memory, multithreaded techniques. In fact, multithreaded approaches that worked on SMPs will work on homogeneous multicores in the exact same manner since memory is shared among all cores in the same way that memory is shared among all processors on an SMP.

Heterogeneous multicore processors have at least one core that is different than the others. Heterogeneity introduces complexity because now either the programmer or a runtime system must determine why and how certain code should run on certain cores. Yet, heterogeneity is desirable for performance reasons. There are two arguments in support of the performance potential of heterogeneity, depending the kind of heterogeneous architecture.

Heterogeneity comes in two flavors: single instruction set architectures (single ISA) and multiple instruction set architectures (multiple ISA). Consider a multicore processor with many simple cores that are identical. Eventually, adding more cores of the same kind will manifest as diminishing performance returns at the application level. This observation is a result of Amdahl's Law: more simple cores will speedup the parallel regions of the application, but the sequential regions of the application will have to run on the same simple cores. Eventually the sequential regions running on the simple cores will become the application's bottleneck. At this point, a larger sequential core that would take up the die space of several simple cores could help overall performance [45]. This approach can also lead to more power efficient processors [63, 93]. This reasoning applies to both single ISA and multiple ISA heterogeneous processors. For single ISA processors, the larger core has the same functionality as the smaller cores, but it has better sequential performance.

Multiple ISA heterogeneous processors open up another opportunity for improved performance. A multiple ISA design can allocate one core to handle running the operating system and interfacing with external devices. The other cores are free to be specialized; if they do not need to run operating system code or interface with other devices, there are fewer constraints on their design. Such cores can be designed specifically for such tasks as dense floating point computations or branch-heavy integer codes. Specialized cores are also free to eschew a traditional memory hierarchy in favor of manual control which has the potential for higher performance.

Recent homogeneous multicore processors have focused on thread level parallelism, but an emerging set of heterogeneous multicores also relies on data parallelism. Architectures such as Larabee [92], the Cell Broadband Engine [23], Pangaea [99] and AMD Fusion [1] merge the CPU and GPU onto the same chip. These are heterogeneous multicore processors; some of the cores have classical designs similar to general purpose processors of the past, while the rest of the cores are specialized vector processors. In this set, the Cell is also a representative of an architecture with an explicitly managed memory hierarchy (EMM). The Cell's vector cores require software management, as they are divorced from the normal memory hierarchy.

By exposing multiple kinds of parallelism, these designs have the potential for high performance. However, the burden for extracting this performance is on programmers more than

before. For sequential processors, both the compiler and the processor itself were responsible for extracting ILP from sequential code. However, multicore processors require that programmers write code with significant thread or data level parallelism. Writing thread and data parallel code has been the domain of high-performance computing for decades. Processor architects have provided novel processor designs, and it falls on the systems and high-performance computing community to explore how to best program these processors.

Currently, writing high performance applications for heterogeneous EMM processors requires an intimate understanding of the underlying hardware and familiarity with the provided API. The APIs for these processors (such as CUDA for GPUs and IBM's Cell SDK) expose architectural details to the programmer. Programming in such environments is analogous to application development in assembly, writing multithreaded programs exclusively with Pthreads, or implementing distributed applications in MPI. All of these methods require knowledge of the underlying mechanisms to produce a working program. Optimizing the program requires extensive architectural knowledge.

Our approach to solve the programmability problem for heterogeneous multicores is to provide a shared memory abstraction. This dissertation presents the design, implementation and performance evaluation of such an abstraction. But, before presenting a shared memory abstraction for heterogeneous multicores, we first discuss parallel programming models in general.

1.1 Parallel Programming Paradigms

There are two dominant parallel programming paradigms: shared memory and message passing. Shared memory parallel programming has a productivity advantage over message passing, although message passing programs have a greater potential for scalability. In this section we introduce these two paradigms in order to motivate why we chose a programming model based on shared memory to solve the productivity and performance challenge of heterogeneous multicores.

Shared memory parallel programming is usually accomplished with some form of multithreading, where multiple threads on the same node share the same address space. Threads can effortlessly communicate because they operate in the same address space, so explicit synchronization must be used to *prevent* communication when it would lead to incorrect results. The dominant basic building block for multithreading on Unix based operating systems is the POSIX Threads (Pthreads) API [54]. Pthreads have various synchronization primitives for coordinating the progress of a computation and memory accesses: mutexes are used to indicate when a thread should have exclusive access to a region of code; condition variables are used for one thread to wait on a signal, and another thread to send that signal; and barriers for when no threads should proceed until all threads have reached the same point.

The main strength of native multithreaded parallel programming—ease of communication

among threads—is also its main weakness. Multithreaded programs are notoriously difficult to code correctly because it is easy to accidentally read or write data without synchronizing; the order and kind of synchronization required even for relatively simple problems can be non-obvious; and even the guarantees from the hardware and software stack are poorly understood by most programmers [8, 73]. While multithreading is powerful, the free-for-all approach of using Pthreads directly is low-level. There are many common tasks in multithreaded programs: maintaining a pool of worker threads, dissembling a larger computation into smaller tasks, distributing these tasks to worker threads, load balancing the work across all worker threads, and synchronizing worker threads.

OpenMP [75, 87] handles all of these common tasks, relieving the programmer from having to contend with them. It is a directive-based extension to C, C++ and Fortran—languages commonly used in high performance computing. It was originally used to parallelize computational loops often seen in scientific applications with heavy use of matrices and vectors. Parallelizing such loops exploits data parallelism. OpenMP has also been extended to include directives to manage task parallelism, which will be less regular than data parallel loops. The gain in productivity is that OpenMP programmers do not have to concern themselves with such details as when or how to create new threads, how to distribute their data among these threads and how to synchronize and initiate the computation itself among the threads. OpenMP's compiler and runtime do the work of mapping sequential code annotated with parallel directives to parallel execution.

Shared memory parallel programming models whose implementations depend on shared memory implemented at the hardware level have a fundamental limitation: they cannot use more than one compute node. They have a fundamental dependence on shared physical memory; there are no mechanisms to communicate with addresses spaces on separate nodes. Message passing paradigms do not have this limitation.

Message passing paradigms assume that the computing infrastructure is composed of multiple nodes with distinct memory address spaces. That is, each compute node can only directly reference its own memory. Communication—of both data and intention—must occur through discrete messages sent from process to process. The de facto standard for distributed computing is the Message Passing Interface (MPI) [82]. MPI abstracts interprocess communication, which may take place on the same compute node, or may take place between different compute nodes across a network. MPI is provided as a library for C, C++ and Fortran, while other languages have provided bindings that call into these standard definitions. There are two main classes of communication types available to MPI programmers: point-to-point communication and collective communication. Point-to-point communication is used for sending or receiving data from one process to another. Collective communications are used when one process communicates with the rest of the processes involved in the computation. Typically, the process initiating collective communication is the master process used to control the progression of the computation. Master processes perform collective operations such as broadcast (sending the same data to all processes), scatter (dispersing elements of a set to all processes), gather (receiving elements of a set from all processes) and

synchronization operations such as a barrier (preventing all processes from proceeding until they reach the same point in the computation).

While MPI does abstract how the inter-process communication occurs (such as over the network or normal inter-process communication on the same node), it does not provide a high-level abstraction for constructing parallel programs. Parallel programs based on message passing have enormous scaling potential—some message passing applications can scale to thousands of cores, with near linear speedup. In practice, however, achieving such speedups is difficult. Programmers must understand the runtime communication patterns of the various components of their application to determine the best communication schedule. Shared memory parallel programming has difficulty scaling to the same degree as message passing because it inherently depends on a single, shared address space. Despite this limitation, implementations of shared memory programming models, such as OpenMP, are popular for their productivity gains [37, 48]. Reasonable investments of programmer effort and time can yield applications that can at least scale with the number of cores on a single node that share an address space.

1.2 Implementing Shared Memory Abstractions on Heterogeneous Multicores

We propose using a shared memory parallel programming model to address the difficulties with programming heterogeneous multicores. However, implementing such an abstraction has many challenges. The processor we use as an example heterogeneous multicore processor is the Cell [23, 53]. The Cell's heterogeneity and the fact that it is an EMM processor present considerable programming difficulties. We first present the salient architectural details of the Cell processor before enumerating the challenges associated with implementing a shared memory parallel programming model on such an architecture.

1.2.1 Cell Specifications

The architecture of the heterogeneous multicore Cell processor is shown in Figure 1.1. One of the cores, the Power Processing Element (PPE), is a 64-bit two-way SMT PowerPC. The other eight cores are 128-bit SIMD-RISC processors called Synergistic Processing Elements (SPEs) and they are typically used as accelerators for data-intensive computation. Each SPE has a 128-bit data path, 128 128-bit registers, and 256 KB of software-managed local store. SPEs can issue two instructions per cycle into two pipelines. One pipeline implements floating point instructions, whereas the other implements branches, load/stores and channel communication instructions.

The SPEs exclusively use code and data from their local stores. They communicate with

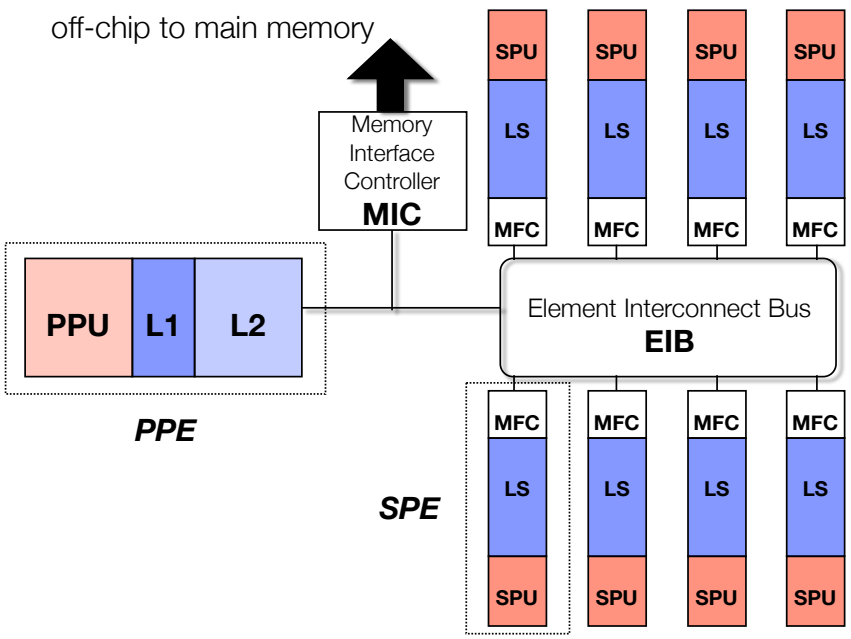


Figure 1.1: Cell architecture. The *PPE* is a typical PowerPC core with hardware managed caches. The *SPEs* have a programmer-controlled *Local Store* which communicates with the *Memory Interface Controller* through their *Memory Flow Controller*.

off-chip memory and with the local stores of other SPEs through Direct Memory Accesses (DMAs). DMAs are posted with channel commands to the Memory Flow Controller (MFC). Each MFC has a queue which can hold up to 16 outstanding DMA requests, each of which can send or receive up to 16 KB of contiguous data. The PPE, SPEs, MIC and I/O controller communicate via the Element Interconnect Bus (EIB), which has a maximum theoretical bandwidth of 204.8 GB/s [9].

1.2.2 Challenges Stemming from Heterogeneity

Implementing a parallel programming model that depends on shared memory is simple when the underlying hardware implements shared memory. On a heterogeneous multicore without such hardware support, a compiler and runtime system that supports shared memory must contend with the following challenges:

Local Memory Spaces Effective use of the Cell requires offloading as much computation as possible to the SPEs. The difficulty this situation presents is that the SPEs are divorced from the normal memory hierarchy. Each SPE has a local storage of 256 KB, and this is the only memory it can directly address. Data must be placed in the local storage explicitly by the programmer. Consequently, programmers must know

the memory access patterns in their application. SPEs can communicate directly with other SPEs, or with main memory through DMAs.

Small Local Storage The 256 KB local storage associated with each SPE contains both all of the code and all of the data the SPE uses. Due to the limited amount of data that can fit into the local storage, streaming data in such a way that communication and computation are overlapped is necessary for good performance. Programmers must be able to anticipate what data they will need for future computations, and initiate DMAs to prefetch this data while other data is in use for computation, or being written back to memory. Overlapping computation and communication can hide the latency associated with DMAs. The relatively large number of outstanding DMAs supported by the MFC further encourages the use of aggressive prefetching and multibuffering techniques for latency overlap.

Further, the small local storage limits the use of recursion. Since the local storage contains all code and all data, the stack frames depth for function calls is severely constrained.

Strided Access A single DMA transfers contiguous data; Cell has no architectural support for strided access to main memory. There are three options for accessing non-contiguous regions of memory:

1. Normal DMAs which transfer the entire contiguous region the strided data resides in. This technique transfers unnecessary data—in the case of large strides, it will transfer a large amount of unnecessary data.
2. Multiple DMAs issued in succession to the MFC. However, the MFC can only have 16 outstanding DMAs, so the 17th DMA will block.
3. Construct DMA lists, and issue a single DMA list instruction to the MFC. Each entry in a DMA list specifies a separate DMA, and it is the programmer's responsibility to ensure that memory addresses for all subsequent entries in the list adhere to the required stride.

In most cases, DMA lists are the most efficient method. Transferring large amounts of unnecessary memory is infeasible for large strides, and the chance of a large number of DMAs blocking is high for multiple DMAs. There is overhead—both in terms of programmer effort and execution time—to constructing the DMA lists. However, this overhead is usually outweighed by the inability of multiple DMAs to scale past 16, and the large amount of unnecessary data single DMAs must transfer.

Data Alignment All DMAs of less than 16 bytes must be *naturally aligned* in both main memory and in the SPE's local storage. That is, transfers of 1, 2, 4 and 8 bytes must be aligned on a 1, 2, 4 or 8 byte boundary respectively. Transfers larger than or equal to 16 bytes must be aligned on a 16 byte boundary—but for best performance, data should be aligned on cache line boundaries (128 bytes).

Such an architecture presents many of the programming difficulties of both message passing and shared memory parallel programming. Using DMAs to transfer data to and from the SPEs is similar to using messages to transfer data in distributed computations. This similarity means that programming the Cell processor shares programming difficulties with distributed computing. But, the DMAs among all of the SPEs are in the same address space. As a consequence of sharing the same address space, many of the same problems from shared memory multithreading are present, such as race conditions and accidentally changing values without the proper synchronization. The Cell presents a "worst of both worlds" scenario to programmers that develop natively for the architecture and system software that seeks to abstract it.

1.2.3 Required Functionality

The preceding challenges imply the functionality that an implementation of shared memory on a heterogeneous architecture must have. Task creation and scheduling exist as requirements when implementing such programming models on shared memory architectures, but the preceding challenges make it more difficult on a heterogeneous platform. The remaining functionality are unique to such an architecture.

Task creation On Cell, the SPEs are independent vector cores with small local memories that execute independently of the main core, the PPE. We must transparently creates these tasks. Being able to execute arbitrary code on the SPEs requires infrastructure that programmers would have to implement by hand for every offloaded region, and for each application. We must eliminate this concern by generating the needed infrastructure.

Parameter passing Transferring parameters to newly created tasks requires knowledge of PPE and SPE communication. We must handle passing parameters to the SPEs upon task creation.

Task scheduling We must determine how to divide user data for parallel execution, and how to distribute that data among the SPEs. We attempt to distribute data as evenly as possible among the SPEs while still adhering to Cell's data alignment constraints.

Buffer management We allocate, deallocate and determine the size of local data buffers for each offloaded region. Local buffers are needed because of the small local storage on the SPEs—the common solution to the limited space is to stream data through the SPEs. By determining the size of the buffers and handling their resource usage, we alleviates programmers from having to consider the SPE's small local storage.

Data communication We must automatically generate Direct Memory Accesses (DMAs) for all shared data. (A DMA is a transfer of data directly to or from main memory.)

We must also determine through static analyses which variables need DMAs from main memory to the SPEs, which need DMAs from the SPEs to main memory and which need both. Further, we need to determine if a series of variable references represent a contiguous or strided access to main memory. For strided accesses, we determine the size of the stride and generates code to handle a non-contiguous memory access.

Results transmission For computations which reduce a computation to a single result, we must handle collecting individuals results from the SPEs and aggregating them into a single result for use by subsequent code on the PPE.

Through automated task scheduling, buffer management and on-demand communication, we are able to provide the abstraction of a single address space among all SPEs. Programmers provide sequential, data-parallel loops with directives indicating which variables will be shared among the SPEs, and our compiler and runtime system generates high performance code that handles all of the above issues. Our shared memory abstraction provides a solution for how to program a heterogeneous multicore processor with reasonable effort while still realizing its performance potential.

1.3 Contributions

At the start of this work, there were many open questions and difficulties that we either aimed to solve, or had to contend with as part of our solutions. The following problems are directly addressed by this dissertation.

- What high-level code representation is suitable for expressing parallelism on a heterogeneous multicore EMM processor?
- What kinds of computations can be expressed in such a programming model and still maintain high performance?
- What level of compiler and runtime support is necessary to enable such computations?
- In terms of performance, how does such compiler and runtime support compare to the usage of a software cache?
- How do programming models with implicit data transfers compare in terms of programmability and performance to programming models with explicit data transfers when programming for heterogeneous multicore processors?
- Given that several kinds of multicore processors are now available, how do they compare to each other with regards to the kinds of computations that they readily support?

It is the job of the high performance computing community to assess the applicability of our current parallel programming techniques to the newly emerging multicore processors. Techniques which abstract the underlying hardware as much as possible are more likely to be adopted by the widest community. Therefore, we have to assess both the programmability of these techniques, and demonstrate their ability to achieve high performance. This dissertation makes the following contributions towards this goal.

- The first published (Schneider et al. [91]) shared memory abstraction for a heterogeneous multicore processor that does not depend on a software cache.
  - An empirical evaluation which demonstrates that with adequate compiler and runtime support, programs written using a high-level shared memory abstraction that hides architectural details can still attain performance comparable to applications written by programmers optimizing specifically for that architecture.
  - Code transformation techniques for replacing references to shared variables with the corresponding references to a local buffer and the data transfers necessary to both implement the abstraction of a shared address space and maintain high performance.
  - Scheduling dense computations at runtime for heterogeneous architectures with restrictive alignment constraints.
  - Supporting stencil code accesses in a shared memory abstraction across heterogeneous cores with separate memory spaces through access analysis at compile time, and a system of rotating buffers at runtime.
  - A simple model for determining an appropriate buffer size for multibuffering that balances using a large buffer with ensuring there are enough transfers to overlap with computation.
  - Performance prediction of computational loops expressed in a shared memory programming model based on a compile-time analysis of the computation's instructions and a runtime data transfer model.
- Empirical evaluation demonstrating the significant performance advantages in implementing a shared memory abstraction for heterogeneous multicores using direct multibuffering instead of a software cache. (Presented in Chapter 4.)
- Quantitative and qualitative comparison of two different programming abstractions for heterogeneous multicores: task based with explicit data division and shared memory with implicit data transfers [91].
- A case study comparing streaming aggregation across three different parallel hardware architectures that represent the current spectrum of available multicores: a homogeneous multicore, a heterogeneous multicore, and a graphics processor [90]. This case study presents the following findings:

- A GPU's connection to main memory is ill-suited for data-movement bound algorithms that perform a single pass through memory.
- A GPU's connection to main memory is also ill-suited for fine-grained memory transfers.
- Starting from similar shared-memory implementations, a compiler which can recognize and generate explicit data transfers for regular memory access patterns can outperform a homogeneous processor that depends on a hardware controlled cache.

These findings point to the importance of future multicore architectures having a low-latency, high bandwidth connection to main memory in order to be able to exploit more kinds of parallelism.

Heterogeneity is likely to be important in the future of computing—high performance computing and even into mainstream programming. The work presented in this dissertation lays the groundwork for how to program such processors in such a way to maintain both productivity and performance.

Chapter 2

Cellgen: Using Shared Memory Abstractions to Program a Heterogeneous Multicore Processor

Cellgen is a compiler that supports a shared memory abstraction on the Cell processor. It started as an attempt to allow PPE and SPE code to live in the same source file. Soon after Cellgen started supporting communication across the PPE-SPE boundary, it became clear the programming model we wanted to support was similar to OpenMP [75]. Offloading data-parallel sections of code to the SPEs naturally fit with the applications we had experience with on Cell.

This chapter provides a brief tutorial for how to program using Cellgen. Cellgen shares some semantics with OpenMP, but legal OpenMP code is not necessarily legal Cellgen code, and vice-versa. This distinction exists because the design approach in Cellgen was to only support memory access patterns that we knew how to generate high performance code for. Over time, Cellgen has been able to support increasingly more kinds of accesses, but it still does not support all access patterns that a full OpenMP implementation does. This chapter presents a brief tutorial of Cellgen, which serves to both provide the reader with an intuitive feel for the programming model and to highlight supported features.

Cellgen is a source-to-source compiler: it accepts C code and emits C code. The current workflow requires a programmer to call cellgen on a $*$.cellgen file, which will produce code for both the PPE and SPE. Currently, we rely on the sophisticated Make files provided by the IBM SDK [52] to produce executable code.

In all of these code examples, we assume the Cellgen blocks of code reside in a legal C program.

The full source code for Cellgen is available at http://www.cs.vt.edu/~scschnei/cellgen.

2.1 Supported Computations

The remainder of this chapter shows what kinds of computations Cellgen supports through example. In this section, however, we explicitly state what kinds of computations Cellgen supports.

Regarding syntax for denoting a Cell region, Cellgen requires the **#pragma** cell directive immediately followed by a set of opening and closing braces to enclose the computation that will execute on the SPEs. Directives for **shared** and **private** variables are not actually required. However, a Cell region that has no set of **shared** variables will not be able to perform any work that will be visible to the rest of the application. While both **shared** and **private** directives can be used to transfer data to the SPEs, only variables in a **shared** set can be used to transfer data out of an SPE. Hence, any interesting Cell region will have at least one variable listed in its **shared** set.

The majority of the programming model semantics is focused on the kinds of accesses that are legal to shared variables. There exists a fundamental distinction between *in* and *out* accesses. An *in* access appears on the right-hand side of an equals sign; it is part of a computation. An *out* access appears on the left-hand side of an equals sign; it is where a computation will be stored. Cellgen is more restrictive with allowable *out* accesses than with *in* accesses because every *out* access is implicitly a write to a location in main memory. Consequently, *out* accesses can only be simple combinations of induction variables. For a single dimensional shared variable named a with induction variable i, the only legal *out* access is a[i]. If a is instead a two-dimensional matrix with induction variables i and j, then the only two legal *out* accesses are a[i][j] and a[j][i]. This pattern continues into the higher dimensions: there is no restriction on the order of accesses, but each individual use of an induction variable must be that induction variable itself. The result of this pattern is that for each iteration (including iterations in nested loops), we are guaranteed that only one iteration will write to one element in the shared variable.

All legal accesses for *out* variables are also legal accesses for *in* variables. The accesses that are legal for only *in* variables are both more interesting and less restrictive. We require that all *out* variable accesses are independent across iterations because of the potential for write-after-write data races. For *in* variables, no such races will happen since we are strictly reading values; there is no harm in reading the same memory location across multiple SPEs. As a consequence, we can relax the independence requirement. This happens in two manners: stencil accesses and unrolled multi-dimensional accesses.

Stencil accesses involve an offset to the induction variable, such as a[i + c] or a[i − c] where c is any constant known at compile time. Stencil accesses are allowed in all dimensions, and can even be used independently—such as a[i − 2][j + 3], or just a[i + 4][j].

Cellgen also supports unrolled multi-dimensional accesses on single dimensional shared variables. Such accesses are in the form of a[i ∗ n + c] where n is a constant known at compile

time and c is less than n. While this access looks like a stencil access, it is semantically the same as a[i][c] assuming that n is the second dimension. The benefit of supporting this special case is that when n is relatively small, the data transfer strategy would transfer many buffers of size n. It is more efficient to transfer buffers that are multiples of n.

All accesses that do not fall into the above categories are illegal. In particular, the set of illegal accesses includes using a function to index a shared variable, such as a[f(i)] or using the values of one shared variable to index another, such as a[b[i]]. The reason that these accesses are illegal is that they are *irregular*; the memory access pattern cannot be determined at compile time based on static code analysis. Cellgen does not support irregular accesses, although it is not impossible for it to do so. The work of Chen et al. [25] could be adapted to work with buffers instead of a software cache.

The final consideration for the legality of an access is scope. Nested loops are allowed in Cellgen (in fact, they are needed for multidimensional array accesses). Each loop must have its own, distinct induction variable. Further, each access to a shared variable must use all of the in-scope induction variables. For example, let a be a two-dimensional matrix, and further assume we have two nested loops with induction variables i and j. In the second nested loop, all of the above legal accesses are allowed. However, if we introduce a third nested loop, it is illegal to access a directly inside of it—assuming that a is a two-dimensional matrix and that valid accesses will involve only i and j. Of course, a can be used in the second-level scope, and its values can be stored in local variables and that local variable can be used in the third-level scope (see Section 2.6 for a concrete example). The reason for this restriction is that every shared variable access is implicitly a request for a data transfer at that level of scope. In the case of a, a value may be needed at the third-level scope, but it does not require a data transfer at that level. It requires the data transfers at the second-level of scope.

2.2 The Basics

All Cellgen code is preceded by a **#pragma** cell directive. Cellgen ignores all other lines of code until it reaches that pragma. The Cellgen code is also enclosed in braces. The simplest Cellgen code transfers no data in or out of the SPE:

```
#pragma cell
{
  printf("Hello world");
}
```

This code will print the string "Hello world" from each SPE. All code within a Cellgen region will be executed on the SPE, and all code outside will be executed on the PPE. In code:

```
printf("I will always execute on the PPE.");

#pragma cell
{
  printf("I will always execute on each SPE.");
}
```

In the previous two examples, the SPEs all behaved the same. While the Cellgen model is to distribute the same code to each SPE, this model is only useful when each SPE operates on separate data. In the following example, each SPE executes different parts of the iteration space for a loop.

```
#pragma cell
{
  int i;
  for (i = 0; i < 10; ++i) {
    printf("iteration %d\n", i);
  }
}
```

In this case, each SPE executes a subset of the iteration space [0–10). We provide further explanation of how each SPE determines its starting and stopping conditions in Section 3.6.

2.3 Computations with Flat Arrays

None of the prior examples performed any interesting computations or even transferred any data beyond loop parameters. The following example multiplies each element of a single-dimensional array by a constant.

```
int vector[SIZE];
int factor; // presumably set elsewhere

#pragma cell shared(int* v = vector) private(int f = factor, int N = SIZE)
{
  int i;
  for (i = 0; i < N; ++i) {
    v[i] = v[i] * f;
  }
}
```

This code sample introduces several new concepts. First, in order to pass data into a Cell region, we must specify if it is **shared** or **private**. Variables declared **shared** will have their data distributed among all SPEs, streamed in or out as needed. Cellgen performs reference analysis to determine how to stream the variables, which we explain fully in Section 3.1. In this example, the data for vector will be both streamed in and out of the SPEs; its result will be visible to code beyond the Cell region. Variables declared **private** will be transferred to each SPE once, and all SPEs will have their own local copy.

Each SPE will carry out its computation in parallel, and there is an implicit barrier at the end of the Cell region. Note that all of the iterations of the loop are *independent*. For variables whose data must be streamed out, it is the programmer's responsibility to ensure that all accesses are independent across iterations. For variables whose data is only streamed in, we can relax this requirement, as shown in the Section 2.7. Cellgen enforces this constraint at compile time.

2.4 Reductions

The result from the previous example was an entire array. Cellgen can also handle reductions, where the computation relies on a large dataset, but the result is reduced to a single value.

```
int vector[SIZE];
int sum = 0;

#pragma cell shared(int* v = vector) reduction(+: int s = sum) private(int N = SIZE)
{
  int i;
  for (i = 0; i < N; ++i) {
    s += v[i];
  }
}
```

After all SPEs have finished, sum contains the summation of all elements of vector. Cellgen supports reductions for addition (+) and multiplication (∗).

2.5 Multidimensional Arrays

Dense matrices are usually implemented with multidimensional arrays in C. Cellgen can handle multidimensional arrays, but it requires more information than with flat arrays, and some programmer assistance is required with column accesses.

To start with, we shall consider row accesses. The following code multiplies each element of a 3-dimensional array by a constant factor:

```
int matrix[N1][N2][N3];
int factor;

#pragma cell shared(int* m = matrix[N1][N2][N3]) private(int f = factor)
{
  int i, j, k;
  for (i = 0; i < N1; ++i) {
    for (j = 0; j < N2; ++j) {
      for (k = 0; k < N3; ++k) {
        m[i][j][k] = m[i][j][k] * f;
      }
    }
  }
}
```

Cellgen needs to know the dimensions of the matrix, which are provided in the **shared** directive. The dimensions can be either constants or variables only known at runtime. Cellgen requires the matrix dimensions so that it can compute addresses for the DMAs which will get and put values in main memory. All of the dimensions of the matrix are implicitly passed as private variables.

Column accesses currently require more work from the programmer. Row accesses access contiguous data in main memory. Data divisions for row accesses are made before the loop starts, and all subsequent accesses are made in contiguous chunks beyond that original division. This data division and access scheme ensures that row accesses are 16-byte aligned by guaranteeing that the original data divisions are 16-byte aligned. Column accesses, however, do not happen in contiguous chunks from a starting address. Rather, they access elements that are separated by a certain stride. For this reason, it is not possible to ensure ahead of time that all subsequent accesses will be correctly aligned. In order to ensure that the column accesses will be legal, Cellgen requires that programmers pad their data. The same computation as the previous example, but accessing columns:

```
typedef struct int16b_t {
  int num;
  char pad[12];
};

int16b_t matrix[N1][N2][N3];
int factor;
```

```
#pragma cell shared(int16b_t* m = matrix[N1][N2][N3]) private(int f = factor)
{
  int i, j, k;
  for (i = 0; i < N2; ++i) {
    for (j = 0; j < N3; ++j) {
      for (k = 0; k < N1; ++k) {
        m[k][i][j].num = m[k][i][j].num * f;
      }
    }
  }
}
```

All subsequent code examples that have accesses that end in .num imply that the programmer must pad that data in the same manner shown above.

2.6 Matrix Multiplication

Consider the simplest expression of a matrix multiplication:

```
int a[N][N]
int b[N][N]
int c[N][N]
int i, j, k;
for (i = 0; i < N; ++i) {
  for (j = 0; j < N; ++j) {
    for (k = 0; k < N; ++k) {
      c[i][j] += a[i][k] * b[k][j];
    }
  }
}
```

Augmenting this exact sequential code with a Cellgen directive violates one of Cellgen's assumptions. The access c[i][j] in the inner-most loop does not use the nearest loop induction variable, k, in its access. If Cellgen allowed such an access, it would generate incorrect DMA accesses for the matrix c because of its placement in the loop. Consequently, such an access is illegal, and Cellgen will flag it as an error. The correct way to express the computation in Cellgen is:

```
#pragma cell shared(int* a = a[N][N], int16b_t* b = b[N][N], int* c = c[N][N])
```

```
{
  int i, j, k;
  for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j) {
      int sum = 0;
      for (k = 0; k < N; ++k) {
        sum += a[i][k] * b[k][j].num;
      }
      c[i][j] = sum;
    }
  }
}
```

We moved the access c[i][j] outside of the inner-most loop so that it only resides in loops whose induction variables it uses. Because Cellgen is an implicit programming model, access placement matters—it uses access placement to infer programmer intentions. Putting a shared variable inside the inner-most loop indicates that how often that shared variable needs to send or receive data is a function of k. Instead, its accesses are only a function of i and j, so we lift it out of the inner-most loop.

2.7 Stencil Accesses

Shared variables that are streamed out—variables that appear on the left-hand side of an equals sign—must be accessed in such a way that each iteration's access is independent. This restriction avoids race conditions.

Shared variables that are exclusively streamed in, however, do not have such race conditions; their values are only read, not written. Hence, we can relax the restriction that their accesses must be independent. Legal accesses are affine expressions involving the induction variable—expressions of the form $ai + b$ where i is an induction variable. The following example demonstrates a Jacobi iteration in Cellgen:

```
#pragma cell shared(double16b_t* a = a[N][N], double16b_t* b = b[N][N])
{
  int i, j;
  for (i = 1; i < N-1; ++i) {
    for (j = 1; j < N-1; ++j) {
      a[i][j].num = (b[i-1][j].num + b[i+1][j].num + b[i][j-1].num + b[i][j+1].num) / 4.0;
    }
  }
}
```

Because `b` is only streamed in, we can relax our requirement that accesses to it must be independent across iterations. Also note that Cellgen supports stencil accesses in two dimensions. This code requires padding because of the code transformations that Cellgen applies to support high performance stencil accesses.

2.8 Summary

Cellgen is an implementation of a shared memory abstraction for the Cell processor. Such an abstraction takes advantage of the fact that the vector cores' DMAs all access the same address space, but protects the programmer from having to contend with the DMAs themselves. However, any abstraction that hides complexities of the Cell's heterogeneity must also preserve performance. If such an abstraction is unable to realize the performance potential of the underlying hardware, then there is no incentive to choose it over more conventional processors.

This chapter demonstrated the kinds of accesses that Cellgen supports: regular array accesses, column accesses of multiple dimensions, and stencil accesses for read-only shared variables. Supporting all kinds of accesses—including irregular accesses—is a goal, but we will only support the kinds of accesses for which we can generate high performance code. In the next chapter, we present in detail the kinds of transformations that we apply to the code presented in this tutorial to achieve high performance on a heterogeneous multicore.

Chapter 3

Code Transformation: Generating High Performance Code for Heterogeneous Multicores

In this chapter we present the transformations which can take high-level code that assumes shared memory and produce code that can execute on a heterogeneous multicore where each of the cores has its own address space. This process requires analyzing how variables are used, classifying variables based on their usage patterns, and then using these classifications to infer the appropriate data transfer strategies. The first classification is analyzing assignments to determine which direction to transfer data. All accesses must also be classified based on how they access main memory: contiguously or with a known stride. Contiguous accesses and strided accesses require generating different kinds of data transfers. A process similar to strip-mining replaces all shared variable references to local buffer references along with the appropriate data transfers. Stencil accesses require special handling both in terms of buffer management and data transfers in order to maximize data reuse. Finally, runtime information is used to determine the best data distribution and buffer size.

Our compiler, Cellgen, targets the Cell processor. These transformations, however, will be necessary on any heterogeneous multicore processor with separate address spaces.

3.1 Reference Analysis

All values passed into accelerated sections are marked either **private** or **shared**, using the same semantics as OpenMP. For the remainder of the section, we trace the transformation of the following computation:

```
#pragma cell private(double factor = f) shared(double16b_t* mtx = m[N1][N2][N3])
{
  int i, j, k;
  for (i = 0; i < N1; i++) {
    for (j = 0; j < N3; j++) {
      for (k = 0; k < N2; k++) {
        mtx[i][k][j].num = mtx[i][k][j].num * factor;
      }
    }
  }
}
```

Private variables are scalars or arrays that are accessed independent of the loop induction variable. Each SPE will see the same value, and will obtain its own local copy.

Shared variables are accessed using the loop induction variable, so their elements must be shared among all of the SPEs. In order to maintain high performance, this data must be streamed through the SPEs. How the variables are used in the accelerated region determines what kind of streaming occurs. During semantic analysis, each time Cellgen encounters a shared variable, it adds it to a set of *in* variables if it occurs on the right-hand side of an equals sign, and to a set of *out* variables if it occurs on the left-hand side of an equals sign. At each scope level (nested loop), Cellgen makes the determination for that scope of the direction of the shared variables used in that scope. If a variable is used exclusively on the right hand side of an assignment, it is transparently classified as an *in* variable by the compiler, which will use double buffering in the generated code to DMA data into the SPE. Cellgen determines the shared variables that appeared exclusively on the right-hand side of an equals sign by subtracting the intersection of the two sets from the *in* set. If a variable is used exclusively on the left hand side of assignment, then it is an *out* variable, and the compiler will use double buffering to DMA data out of the SPE back to main memory. Cellgen makes this determination by subtracting the intersection of the two sets from the *out* set. Variables that appear on both sides of an assignment are classified as *inout* variables, which are triple buffered to DMA data in and out of the SPE. Cellgen knows which variables are *inout* because that is the intersection of the two sets.

The buffering principle is that a local buffer should exist for each state that a variable can be in. *In* variables can be in two states: transferring data in, and computing. *Out* variables can also be in two states: computing and transferring data out. *Inout* variables can be in three states: transferring data in, computing, and transferring data out. Maintaining variables in as many states as possible allows us to overlap computation and communication. This technique uses more memory in the SPEs local store, but it allows for significant performance gains by preventing computations from waiting on data from main memory.

In the prior example, Cellgen recognizes that `mtx` is used on both sides of an assignment, and classifies it as an *inout* variable.

3.2 Access Analysis

DMAs on the Cell can transfer only contiguous memory. Code on an SPE that needs to access noncontiguous regions of memory—such as the columns of a matrix stored in row-major format—must be handled differently than code that simply accesses contiguous regions.

Cellgen recognizes noncontiguous multidimensional accesses by comparing the order of the accesses with the most closely nested loop induction variable—the one changing the fastest. If the nearest induction variable and the right-most access match, then the access pattern is contiguous, and a single DMA can be used to transfer data. However, if the accesses and induction variable don't match, then Cellgen generates code to transfer noncontiguous data.

As explain in Section 1.2.2, noncontiguous data must be transferred with DMA lists on the Cell. The lists are a special data structure in the SPE's local storage that contains a list of addresses and sizes that should be transferred. Cellgen will generate these lists and issue the DMA to the MFC for all column accesses that it encounters.

In the example we are following throughout this section, Cellgen recognizes that the closest induction variable to mtx's access is k, which is the middle access. Since mtx is stored in row-major format, Cellgen recognizes this access as a column access and will generate DMA lists.

3.3 Buffer Substitution

During code generation for the SPE, all private and shared variables need their accesses replaced with a local buffer access. For private variables, this is trivial since, by definition, their value does not change during a loop. Shared variables require special handling.

The simplest technique for handling shared variables is to map the induction variable space to the buffer space, as shown in the following example:

```
1  doublt16b_t *mtx_buf = malloc_align(sizeof(doublt16b_t) * buf_sz);
2  dma_list mtx_lst;
3  int i, j, k;
4  for (i = spe_start; i < spe_stop; i++) {
5    for (j = 0; j < N3; j++) {
6      for (k = 0; k < N2; k++) {
7        if (!(k % buf_sz)) {
8          void* adr = mtx_adr + ((i * N2 + k) * N3) + j;
9          int stride = N3 * sizeof(double16b_t);
10         mtx_lst = construct_dma_list(adr, buf_sz, stride);
```

```
11          dma_list_in(mtx_buf, mtx_lst, adr);
12        }
13
14        // actual computation
15        mtx_buf[k % buf_sz] = mtx_buf[k % buf_sz] * factor;
16
17        if (!((k+1) % buf_sz)) {
18          void* adr = mtx_adr + ((i* N2 + k + 1 - buf_sz) * N3) + j;
19          int stride = N3 * sizeof(double16b_t);
20          mtx_lst = construct_dma_list(adr, buf_sz, stride);
21          dma_list_out(mtx_buf, mtx_lst, adr);
22        }
23      }
24    }
25  }
```

This code example does not use multibuffering to simplify its presentation. The presented code represents a technique which trivially handles variables with different buffer sizes. However, the mapping from iteration space to buffer space requires a modulus operation for every shared variable access. The overhead from the modulus operation can be substantial.

A better technique is similar to strip-mining [1], as shown in the following:

```
1  doublt16b_t *mtx_buf = malloc_align(sizeof(doublt16b_t) * buf_sz);
2  dma_list mtx_lst;
3  int i, j, k;
4  for (i = spe_start; i < spe_stop; i++) {
5    for (j = 0; j < N3; j++) {
6      int stop = N2 - (N2 % buf_sz);
7      for (k = 0; k < stop; k += buf_sz) {
8        void* adr = mtx_adr + ((i * N2 + k) * N3) + j;
9        int stride = N3 * sizeof(double16b_t);
10       mtx_lst = construct_dma_list(adr, buf_sz, stride);
11       dma_list_in(mtx_buf, mtx_lst, adr);
12
13       int ___k;
14       for (___k = 0; ___k < buf_sz; ___k++) {
15         mtx_buf[___k] = mtx_buf[___k] * factor;
16       }
17
18       adr = mtx_adr + ((i * N2 + k - buf_sz) * N3) + j;
```

[1] Strip-mining is a compiler optimization technique which splits one loop into two nested loops. The nested loop iterates over a fixed size. Strip-mining itself is not actually an optimization, but it can enable other optimizations.

```
19          mtx_lst = construct_dma_list(adr, buf_sz, stride);
20          dma_list_out(mtx_buf, mtx_lst, adr);
21        }
22      }
23    }
```

The modulus operation is avoided by introducing an additional loop. Since the SPE does not have a dynamic branch predictor, it is usually best to avoid branches. However, the SPE does have instructions for software hints that can reduce the cost of a correctly hinted branch to a single cycle. Regular **for** loops such as the ones generated by Cellgen are amenable to such software hints. Consequently, the extra branches are significantly less expensive than the modulus operations.

An additional benefit is that the navie version also requires modulus operations and a branch to determine if data should be streamed in or out (lines 7–12 and 17–22 in the naive version). By adding an additional loop that consumes an entire buffer, we guarantee that when execution reaches the top or the bottom of the original loop, data is ready to be sent or received (lines 8–11 and 18–20 in the strip-mined version). The benefit from this difference is not requiring an expensive modulus operation, and not having a branch inside the computation loop.

3.4 Multibuffering

The prior examples did not include multibuffering in order to highlight the overall buffer substitution strategy. Multibuffering, however, is the means by which Cellgen is able to overlap computation with communication. Hence, the technique is integral to achieving high performance.

In this example, mtx is an *inout* shared variable which means that at any given time, an SPE can be fetching data for it, performing computations on it, and sending results back to main memory. Cellgen maintains three buffers for each of these states. In the code below, the variables prev and mtx_nxt are used to keep track of buffer state:

```
1  #define INOUT 3
2  doublt16b_t (*mtx_buf)[buf_sz] = malloc_align(sizeof(doublt16b_t) * buf_sz * INOUT);
3  double16b_t* mtx;
4  dma_list mtx_lst[INOUT];
5  int prev, mtx_nxt = 0;
6  void* adr;
7  int stride;
8
```

```
 9  int i, j, k;
10  for (i = spe_start; i < spe_stop; i++) {
11    for (j = 0; j < N3; j++) {
12      adr = mtx_adr + ((i * N2) * N3 + j);
13      stride = N3 * sizeof(double16b_t);
14      mtx_lst[mtx_nxt] = construct_dma_list(adr, buf_sz, stride);
15      dma_list_in(&mtx_buf[mtx_nxt], mtx_lst[mtx_nxt], adr);
16
17      for (k = 0; k < N2; k += buf_sz) {
18        prev = mtx_nxt;
19        mtx_nxt = (mtx_nxt + 1) % INOUT;
20
21        dma_wait(mtx_nxt);
22
23        adr = mtx_adr + ((i * N2 + (k + buf_sz)) * N3 + j);
24        mtx_lst[mtx_nxt] = construct_dma_list(adr, buf_sz, stride);
25        dma_list_in(&mtx_buf[mtx_nxt], mtx_lst[mtx_nxt], adr);
26
27        dma_wait(prev);
28        mtx = &mtx_buf[prev];
29
30        unsigned int __k = 0;
31        for (__k = 0; __k < buf_sz; __k++) {
32          mtx[__k].num = mtx[__k].num * factor;
33        }
34
35        adr = mtx_adr + ((i * N2 + k) * N3 + j);
36        dma_list_out(mtx, mtx_lst[prev], adr);
37      }
38    }
39  }
```

Before the inner-most loop, lines 12–15 perform the first data fetch for the shared variable. This transfer cannot be overlapped with computation. Once inside the inner-most loop, we must first save the old buffer state value (line 18), then determine which buffer is next (line 19). The wait on line 21 ensures that before we continue, the last sending of results has completed. We must do this because lines 23–25 will overwrite those locations in the local store with data from main memory for the next iteration. The wait on line 27 ensures that all of the data for this iteration has finished transferring, and lines 30–33 perform this iteration's computation. Finally, lines 35–36 write the results from this iteration's computation back to main memory. Both of the calls to dma_wait() (lines 21 and 27) will block if the data transfer associated with that tag has not yet completed.

3.5 Remainders

All of the previously presented strip-mined examples have an implicit assumption: that the amount of data in the inner-most loop is a multiple of the buffer size. This assumption will not hold in the general case, so Cellgen must handle leftover data. The following code is a complete transformation of the original Cellgen example:

```
1  #define INOUT 3
2  doublt16b_t (*mtx_buf)[buf_sz] = malloc_align(sizeof(doublt16b_t) * buf_sz * INOUT);
3  double16b_t* mtx;
4  dma_list mtx_lst[INOUT];
5  int prev, mtx_nxt = 0;
6  void* adr;
7  int stride;
8
9  int i, j, k;
10 for (i = spe_start; i < spe_stop; i++) {
11   for (j = 0; j < N3; j++) {
12     adr = mtx_adr + ((i * N2) * N3 + j);
13     stride = N3 * sizeof(double16b_t);
14     mtx_lst[mtx_nxt] = construct_dma_list(adr, buf_sz, stride);
15     dma_list_in(&mtx_buf[mtx_nxt], mtx_lst[mtx_nxt], adr);
16
17     int mtx_rem = N2 % buf_sz;
18     int mtx_ful = N2 - mtx_rem;
19
20     for (k = 0; k < mtx_ful; k += buf_sz) {
21       prev = mtx_nxt;
22       mtx_nxt = (mtx_nxt + 1) % INOUT;
23
24       dma_wait(mtx_nxt);
25
26       adr = mtx_adr + ((i * N2 + (k + buf_sz)) * N3 + j);
27       mtx_lst[mtx_nxt] = construct_dma_list(adr, buf_sz, stride);
28       dma_list_in(&mtx_buf[mtx_nxt], mtx_lst[mtx_nxt], adr);
29
30       dma_wait(prev);
31       mtx = &mtx_buf[prev];
32
33       unsigned int __k = 0;
34       for (__k = 0; __k < buf_sz; __k++) {
35         mtx[__k].num = mtx[__k].num * factor;
36       }
```

```
37
38          adr = mtx_adr + ((i * N2 + k) * N3 + j);
39          dma_list_out(mtx, mtx_lst[prev], adr);
40        }
41
42        if (mtx_rem) {
43          dma_wait(mtx_nxt);
44          mtx = &mtx_buf[mtx_nxt];
45
46          unsigned int ___k = 0;
47          for (___k = 0; ___k < mtx_rem; ___k++) {
48            mtx[___k].num = mtx[___k].num * factor;
49          }
50
51          adr = mtx_adr + ((i * N2 + mtx_ful) * N3 + j);
52          dma_list_out(mtx, mtx_lst[mtx_nxt], adr);
53
54          dma_wait(mtx_nxt);
55        }
56      }
57    }
```

On lines 17 and 18, we use the original bounds of the inner-most loop and the buffer size to determine both how many full iterations there are, and how many elements, if any, are leftover. The stopping condition on line 20 uses the number of full iterations. The remainders are handled starting on line 42. First, we must wait for the last chunk of data to complete transferring, which is done on line 43. The final computation, over just the remaining data, occurs on lines 46–49. Finally, the remainder results are transferred back to main memory on lines 51 and 52.

3.6 Scheduling

Cellgen distributes the outermost loop iterations to the SPEs to run in parallel. Currently, we have implemented only static scheduling, which attempts to distribute the iterations to SPEs as evenly as possible to avoid imbalances. Each SPE calls compute_bounds to determine its place in the overall computation:

```
void bounds_assign(int* start, int* stop, const int cutoff_id,
  const int bytes16, const int base_chunks, const int leftover)
{
  *start = *start + (spe_id * base_chunks * bytes16);
```

```
  if (spe_id > cutoff_id) {
    *start += bytes16 * (spe_id - cutoff_id));
  }

  if (spe_id >= cutoff_id) {
    *stop = *start + ((base_chunks + 1) * bytes16 + leftover);
  }
  else {
    *stop = *start + (base_chunks * bytes16 + leftover);
  }
}

void compute_bounds(int *start, int *stop, size_t element_sz)
{
  const int bytes16 = 16 / element_sz;
  const int total_chunks = (*stop - *start) / bytes16;
  const int base_chunks = total_chunks / spe_threads;
  const int thread_bytes_rem = ((*stop - *start) % (bytes16 * spe_threads));
  const int leftover = thread_bytes_rem % bytes16;
  const int cutoff_id = spe_threads - (thread_bytes_rem / bytes16);

  if (spe_id == spe_threads - 1) {
    bounds_assign(start, stop, cutoff_id, bytes16, base_chunks, leftover);
  }
  else {
    bounds_assign(start, stop, cutoff_id, bytes16, base_chunks, 0);
  }
}
```

Each SPE is sent the original starting and stopping conditions of the loop. That information, as well as the size of the smallest data type its accelerated region operates on, are passed to compute_bounds. First, it determines how many of the elements are contained in 16-bytes (bytes16, line 20), and it uses that value to determine how many 16-byte chunks are contained in the entire iteration space (thread_chunks, line 21). The total number of active SPEs is always kept in spe_threads, which is used to determine the base number of iterations each SPE should receive (base_chunks, line 22). Next, it needs to determine the remaining iterations that it could not evenly distribute among the SPEs, and must be added to the last SPE (leftover, line 24). This adjustment causes a further adjustment to the starting condition of other SPEs with an ID higher than cutoff_id.

This is perhaps a surprising amount of computations to perform a fundamentally simple task: evenly divide the iteration space. But these computations must be performed because an SPE with too many iterations will cause a work imbalance and lengthen the runtime of

```
#pragma cell shared(double* in = in[N][N], \
                    double* out = out[N][N])
{
  int i, j, k;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      out[i][j] = in[i][j-1] + in[i][j] + in[i][j+1];
    }
  }
}
```

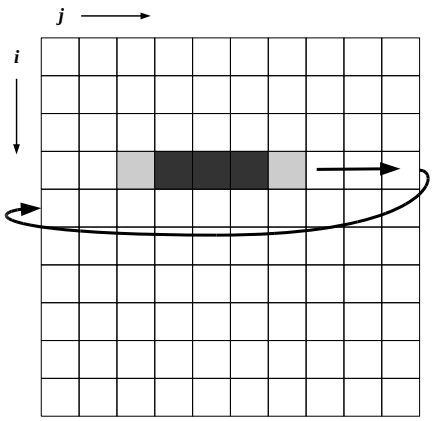


Figure 3.1: Code and diagram for on-induction stencil access. We call this code such because j is the fastest-changing induction variable and the stencil access is relative to it. The darkly shaded blocks of the diagram mark the (i, j) iterations that are a part of the buffer. The lightly shaded blocks are the extra elements that need to be fetched because of the stencil accesses at the edge of the buffer's iteration space.

the entire offloaded region. However, we cannot simply divide the iteration space evenly among all SPEs. Each iteration is implicitly also an address in main memory; dividing the iteration space up without respecting the size of the data types could result in a misaligned DMA, which causes a bus error.

3.7 Stencil Analysis

The final transformation is to accommodate stencil accesses for *in* variables. Cellgen supports 1- and 2-dimensional stencil accesses, which we refer to as *on-induction* and *off-induction* accesses for reasons we will explain in this section. The purpose of our stencil optimizations are to reduce the number of DMAs required for a computational kernel by intelligently reusing data that is already in the SPE's local store.

3.7.1 On-induction Stencil Access

Figure 3.1 presents a code example for on-induction stencil access and an accompanying diagram which shows how Cellgen handles the access. Stencil accesses that are on the same dimension as the fastest changing induction variable—the induction variable that controls the inner-most loop—are comparatively easier to handle than off-induction accesses.

Cellgen analyzes each shared-array access and keeps track of three quantities for each shared

```
#pragma cell shared(double* in = in[N][N], \
                    double* out = out[N][N])
{
  int i, j, k;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      out[i][j] = in[i-1][j-1] + in[i+1][j] +
                  in[i+2][j+1];
    }
  }
}
```

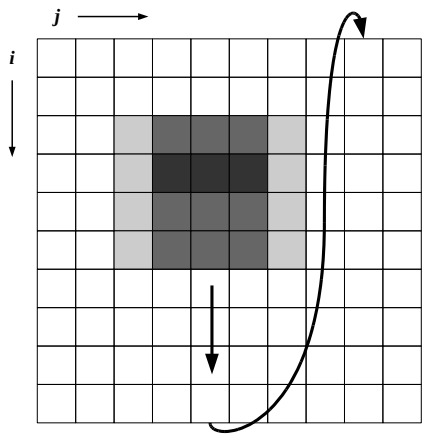


Figure 3.2: Code and diagram for on- and off-induction stencil access. In this case, j is the fastest-changing induction variable, but there are also stencil accesses relative to i. Note that the iteration space has been transformed. The duplicate, medium-shaded blocks represent both past and future iterations in the i dimension.

variable: the lowest stencil access, the highest stencil access, and the *spread*, which is the absolute value of the highest minus the lowest access. For example, in Figure 3.1, the lowest access is -1, the highest access is 1, and the spread is $|1-(-1)| = 2$. All of these values are used to calculate how much extra data to fetch for each buffer, and the relevant addresses in main memory of that data.

As shown in the diagram for Figure 3.1, Cellgen is able to handle on-induction stencil accesses by fetching more data at the beginning of a buffer (to accommodate the lowest stencil accesses by the first iterations in the buffer), and more data at the end of the buffer (to accommodate the highest stencil accesses by the last iterations in the buffer). Cellgen accomplishes this stencil support without making changes to the structure of the code; rather, it makes adjustments to the amount of data transferred based on the analysis of the shared variable's stencil accesses.

3.7.2 Off-induction Stencil Access

We define off-induction stencil accesses to be stencil accesses that do not use the fastest-changing (lexically closest) induction variable. Figure 3.2 adds off-induction stencil accesses to the code from Figure 3.1. The key insight to efficiently handling stencil accesses is to reuse data that is already in the SPE's local store. For on-induction accesses, this reuse is simply a matter of fetching a little more data for each DMA (of course, exactly *what* data to fetch is important and was covered in the previous section).

However, for off-induction stencil accesses, the reuse must occur at the level of an entire buffer. This fact implies two requirements: Cellgen must have a scheme for handling a rotating system of buffers beyond what is already needed for multibuffering, and the iteration space must be transformed such that such a rotating system of buffer reuse is possible.

Cellgen still has to keep track of buffers used for the current computation and buffers to prefetch data for the next computation. However, for off-induction stencil accesses, the number of buffers for the current computation is equal to the spread of the off-induction variable. Cellgen determines the spread at compile time in the same way it does so for on-induction stencil accesses. This rotating systems of buffers allows the same data to step through each of the roles as the computation advances. For example, consider the code in Figure 3.2. When the outer loop is on iteration i, the inner loop requires data from rows $i-1$, $i+1$ and $i+2$. Let us assume that i is safely in the middle of the iteration space, but we have not yet started its computation. This assumption implies that the data for rows $i-1$, $i+1$ and i are already in the SPE's local store. (While data from row i is not needed to compute any values at iteration i, we needed the data in a previous iteration and we will need it again.) The only data that is not already in the SPE's local store is that for row $i+2$, and that is the only data that we need to fetch for this iteration.

Once iteration i is complete, Cellgen still needs the data fetched for row $i+2$. Cellgen rotates the data from row $i+2$ into the position that the data from row $i+1$ previously occupied, it rotates the data from row $i+1$ into the position that row i previously occupied, on down to the lowest row, which Cellgen drops because it is no longer needed. The rotations are cheap: they only require computing a local address and assigning it to a pointer. Cellgen generates the exact rotations needed at compile time, based on the accesses seen in the code.

This system of rotating buffers ensures that for all of the non-border iterations, only a single DMA is needed, just as in the non-stencil case. Once we add one more buffer, we can prefetch part of a row for a future iteration while working on the current iteration. Double and triple buffering as described in Section 3.1 becomes N buffering where N is the spread of the off-induction variable.

In order for this scheme to work, Cellgen must ensure that each successive iteration in the inner loop operates on a new row. If the iteration space as provided by the user does not follow this pattern, Cellgen transforms the iteration space so that it does. If Cellgen did not perform this transformation, then the system of buffers would contain data from the same row as the current iteration—which does not help when the induction access is across rows.

As an illustration of this transformation, consider the code and iteration space in Figure 3.1. In this code, j is the fastest changing induction variable, and as a result, the direction of the computation in the diagram proceeds in the j dimension. In Figure 3.2, j is still the fastest changing induction variable. However, Cellgen transforms the iteration space, as shown in the diagram. In Figure 3.1, each SPE operates on a set of rows, while in Figure 3.2, each SPE operates on a set of columns.

Finally, as implied by Figure 3.1, on- and off-induction stencil accesses can work in conjunction with each other. The transformations they require are independent of each other. Currently, Cellgen can only handle stencil accesses in two-dimensions. Generalizing this scheme to work with an arbitrary number of dimensions is future work.

3.8 Buffer Size

Obtaining high performance on Cell requires overlapping computation and communication so that communication latency is hidden. This is the purpose of multibuffering, and the reason why Cellgen determines which kind to use. In order to hide latency, the computation and communication must take the same amount of time. If computation is too short compared to communication, SPEs will waste time waiting on DMAs. The computation and communication time are determined by the buffer size: the larger the buffer size, the longer each DMA will take, but there's also more computation to do while waiting. Consequently, Cellgen's strategy is to pick the largest buffer size that ensures at least some of the computations will be overlapped.

Cellgen sets the buffer size as:

$$B = \min\left(\frac{N_b}{d+1}, 16384\right) \tag{3.1}$$

where N_b is the number of bytes needed for a variable in the nearest loop, d is the depth of the buffer (either 2 or 3 for double or triple buffering), and 16384 bytes is the maximum amount of data that the Cell can transfer in one DMA. The actual buffer size decision is made at runtime by Cellgen's generated code.

The rationale behind Equation 3.1 is that, as our experiments in Chapter 4 show, the crossover point where DMA latency is hidden by computation happens early, with relatively small buffer sizes. After this point, as the buffer size increases, so does the amount of work the SPE can do between transfers, keeping performance constant. However, naively choosing the largest possible buffer size for sizes of N_b less than the maximum possible DMA transfer size eliminates the possibility of computation and communication overlap. Dividing N_b by $d+1$, one more than the buffer depth, ensures that the shared variable is in as many states as possible. The quantity $d+1$ is used instead of just d because there always must be an initial, priming DMA before DMA transfers and SPE computation overlap.

Equation 3.1 represents an upper bound on the buffer size. We err on the side of making the buffers too large because our experience shows us that buffers that are too small are more likely to cause a performance problem.

We consider the application of Equation 3.1 dynamic buffer selection because buffer choice is determined at runtime by N_b. Users also have the option to provide a static buffer selection,

which will override the dynamic option.

3.9 Performance Prediction

Cellgen provides a prediction of how long an accelerated region will take to complete. We plan to use this prediction in the future to make scheduling decisions, particularly how the PPE should behave after an offload. In this dissertation, we only present the prediction and evaluate its accuracy. In future work we plan to apply this prediction scheme to dynamic scheduling of computation and data transfers. Our prediction scheme is based on both a static analysis of the code and runtime information. The prediction is provided in the form:

$$T_{offload} = \max(T_{DMA}, T_{comp}) \tag{3.2}$$

Where T_{DMA} is the estimated time spent on DMAs and T_{comp} is the estimated time spent on computation. Both T_{DMA} and T_{comp} depend on N_b, the total amount of bytes transferred, and N_{SPE}, the total number of SPEs used. These values are known only at runtime. This model assumes that communication and computation are overlapped as much as possible, which is why the total time for an offload is whichever time dominated.

3.9.1 Data Transfer Model Derivation

We estimate the time spent on data transfers using:

$$T_{DMA} = \frac{S + L(N_b)}{N_{SPE}} \tag{3.3}$$

In this case, S is the startup cost of a DMA, which includes the cycles needed by the MFC to prepare the transfer, and the overhead of the code generated by Cellgen. We measure S to be 128 cycles. $L(N_b)$ is a linear model for the cycle cost of the total amount of bytes to be transferred. We derive a linear model of data transfer latency as:

$$L(N_b) = \begin{cases} 349.70 + 0.13N_b & \text{if } 16 \leq N_b \leq 2\text{k}, \\ 472.76 + 0.16N_b & \text{if } 2\text{k} < N_b \leq 4\text{k}, \\ 306.45 + 0.21N_b & \text{if } 4\text{k} < N_b < 16\text{k} \end{cases} \tag{3.4}$$

We derived $L(N_b)$ through an experiment which measures DMA latency as we varied the transfer size by 128 bytes. The benchmark uses a single SPE, and performs 1 million DMA get operations of the specified size. We vary the transfer size with a granularity of 128 bytes because that is the size of a cache line and the minimum data transfer unit on the Cell ring

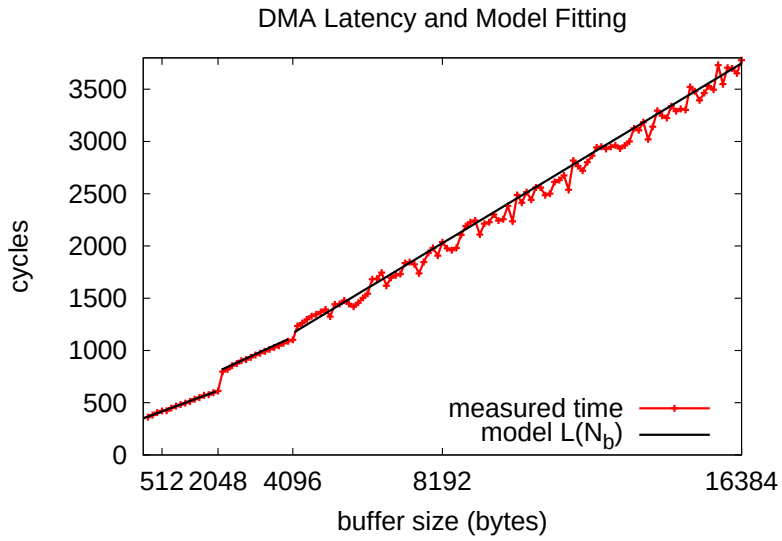


Figure 3.3: DMA latency as a function of transfer size.

network. The data points labeled "measured time" in Figure 3.3 are average cycle counts from each of these runs. We recognize that these cycle counts will skew slightly higher because they include the cost of measuring.

The measured latency is clearly divided in three segments: 128 bytes to 2048 bytes, greater than 2048 bytes to 4096 bytes, and everything greater than 4096 bytes. Because of this clear division, which reflects overhead for loading the SPE translation table, we performed a separate linear regression for each section, which gives us Equation 3.4. This equation is plotted on Figure 3.3 as "model $L(N_b)$."

From this same experiment we also determined the startup cost of a DMA as 128 cycles. The MFC itself only takes 16 cycles to issue a DMA, but we are also measuring the address calculation, the function call cost to our own library, and the code executed in the function before issuing the DMA instruction to the MFC.

This model ignores several aspects of Cell. It assumes that all data is transferred at once, without any buffering. It also ignores effects such as interference from other SPEs, the costs associated with accessing the memory module itself, and models all DMAs as the more expensive DMA get. However, we have purposely used a simplified model that is similar to how Cellgen generates code, and is inexpensive to compute at runtime for each accelerated region.

Table 3.1: Cycle counts used to estimate the cost of a computation.

	+	-	*	/	%	load	store
char	2	2	7	7	7	2	6
int	2	2	7	7	7	6	6
float	6	6	6	6	6	6	6
double	13	13	13	13	13	6	6

3.9.2 Computation Model Derivation

Cellgen determines the value for T_{comp} for each accelerated region through static code analysis. This model assumes that the computational kernel in an offloaded region is a basic block: no branches, including function calls.

Since the contents of the local storage are controlled by software, statically determining the runtime cost of a block of code is significantly easier than on a conventional architecture. Unlike on a conventional architecture, there is no concept of a cache miss. The model in the previous section already accounts for data transfer costs, and the overall model assumes that computation and communication are overlapped. Consequently, we should be able to estimate the cost of an iteration by analyzing the specific operations and their types in an iteration. The estimate is defined as:

$$T_{comp} = \frac{N_b \max(T_0, T_1)}{zN_{SPE}} \tag{3.5}$$

Where T_0 is the amount of time spent on pipeline 0, T_1 is the amount of time spent on pipeline 1, and z is the size of an element in bytes. The SPEs have a dual issue pipeline. All loads and stores are issued on pipeline 0, and all computation is issued on pipeline 1. Since the SPEs execute instructions on the two pipelines in parallel, we take the maximum of the two cycle counts, not the addition.

During reference and access analysis, Cellgen keeps track of the number and kinds of operations it encounters, as well as the types involved. It is also aware of common optimizations. For example, a multiplication followed by an addition will use a single multiply-and-add instruction instead of two separate instructions. The process of reference and access analysis generates code transformations that will add some overhead. Once analysis is complete, Cellgen also counts operations in its own generated code and factors them into the model as part of the cost of an iteration, or as part of the startup cost of a DMA.

The cycle counts Cellgen uses are presented in Table 3.1, which come from The Cell Programming Handbook [53]. When Cellgen encounters any of these operations, it determines the type of the expression using the declared type of the variables in use, and through type inference of subexpressions using C's type promotion rules.

3.10 Summary

In this chapter we presented the code transformations that Cellgen applies to a programmer's high-level code in order for it to execute in parallel on multiple SPEs. Such transformations are necessary for implementing a shared memory abstraction for any heterogeneous processor which has cores divorced from the main memory hierarchy. However, some of the transformations can be adapted for cores that have a hardware controlled cache, but allow for software prefetching hints. Buffer management would no longer be necessary, but the same schedule for fetching data from main memory overlapped with computation could be applied.

In the next chapter, we evaluate the performance of these transformations and runtime decisions.

Chapter 4

Experimental Evaluation: Measuring the Performance of Code Transformations for Heterogeneous Multicores

In this chapter we evaluate the prior claims that the transformations we apply to code that assumes shared memory generates code that performs well on a heterogeneous multicore. We also evaluate the individual components and features of Cellgen, such as dynamic buffer sizing, modelling accuracy and stencil support.

Cellgen supports only memory access patterns for which we know how to generate high performance code for. An alternative approach to supporting a shared memory abstraction on a heterogeneous multicore with cores that have separate address spaces is to implement a software cache. In a software cache, all shared variable references become requests into a software controlled cache which fetches chunks of data from main memory on a demand basis. All shared variable accesses must go through the software cache, which increases the execution time even if the data is present in the cache. Such an approach also prevents overlapping computation with communication; accesses must be known in advance in order to do so. Using dense matrix and vector computations, we demonstrate the effectives of recognizing access patterns at compile time, and generating high performance multibuffering code instead of relying on a software cache.

Using two real applications, we also test the buffer size selection which is made at runtime, and evaluate the accuracy of our performance prediction. Finally, we also evaluate the effectiveness of our approach to handling stencil accesses in two different ways. The first evaluation determines how much time is saved versus using a naive method which uses a separate buffer for each access. The second evaluation compares the performance of code generated by Cellgen using the techniques presented in the previous chapter against an

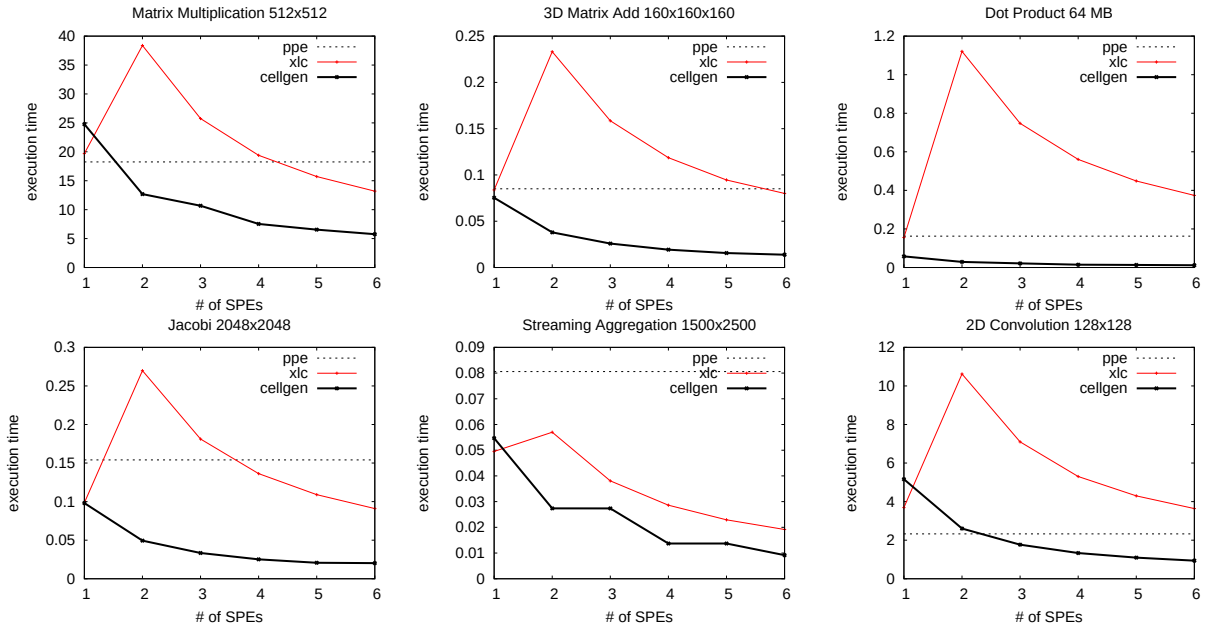


Figure 4.1: Microbenchmarks comparing Cellgen and the OpenMP implementation in XLC, which is a part of the Cell SDK.

application written by an expert programmer using the Cell SDK directly.

All of the experiments in this chapter use an experimental platform consisting of a Sony PlayStation 3 running Fedora 7, using Linux kernel 2.6.24 and IBM's Cell SDK 3.0. On a PS3 running Linux, only 6 SPEs are available to the programmer.

4.1 Microbenchmarks

We evaluate Cellgen's ability to scale as the number of SPEs increase with six data-parallel, computational kernels. We also compare Cellgen's performance with that of the implementation of OpenMP in IBM's XLC C and C++ compiler, version 9.0. The design of this implementation is described by Eichenberger et al. [32, 33]; it relies on a software cache with limited support for direct buffering. Consequently, these experiments also demonstrate the limitations of relying on a software cache for dense computations on the SPEs. The Direct Block Data Buffer (DBDB) approach described by Liu et al. [72] is *not* a part of this version of XLC. Cellgen and DBDB have similar approaches to supporting shared memory across SPEs; see Section 7.1 for more discussion of their relationship.

Descriptions of the microbenchmarks:

Matrix Multiplication: Multiplies two 2-dimensional matrices of 512 × 512 **double** floating point numbers. The matrices being multiplied are exclusively *in* values and their

result is exclusively an *out* value.

3D Matrix Add: Adds two 3-dimensional matrices of 160 × 160 × 160 **double** floating point numbers. As before, the matrices being added are exclusively *in* values and their result is exclusively an *out* value.

Dot Product: Performs a dot product across two 64 MB vectors of **double**s. The vectors are input, and the result is a scalar reduction.

Jacobi: Similar to the implementation described in Section 4.6.2. Performs a Jacobi computation on two 2-dimensional matrices of 2048 × 2048 **double**s. Note that this benchmark takes advantage of the stencil optimizations described in Section 3.7.

Streaming Aggregation: The same operation as described in Section 6.2.2 with source code in Figure 6.5. Performs a parallel aggregation on integers in a table with 1500 groups and windows of size 2500.

2D Convolution: Performs a 2-dimensional convolution on an input matrix of 255 × 255 integers and an input matrix of 128 × 128, with a result of 128 × 128 integers. The operation loops over every element of the result matrix, performing a convolution at each result element using the input matrices. This algorithm is N^4, where N is the dimension of the result matrix.

As shown in Figure 4.1, Cellgen consistently scales as the number of SPEs increases. For each microbenchmark, however, XLC's performance decreases by a factor of 1.9–11.5 when 2 SPEs are used compared to just a single SPE. Using only one SPE is a special case for the software cache used by XLC. When multiple SPEs are used, the caching system keeps track of which bytes on a cache line have been written to support multiple writers across the SPEs. When only one SPE is used, however, it does not need to keep track of byte-level writes.

The Dot Product microbenchmark is unique in two ways: it is the only benchmark that has a reduction on a scalar, and it has the largest performance gap at 6 SPEs between XLC and Cellgen: Cellgen is over 31 times faster. We attribute this performance gap to the differences in how the two systems handle the reduction. For XLC, the cache line containing the scalar variable is likely bouncing between the SPEs. Cellgen, on the other hand, uses a local value in each SPE to store the reduction for just the computations on that SPE, and does not synchronize with the other SPEs until after all of them have finished.

For all of the microbenchmarks (except Dot Product), when using all 6 SPEs, Cellgen is 2–5.8 times faster than XLC's implementation of OpenMP. From these results, we draw the conclusion that naive software caches are at a serious performance disadvantage for supporting a shared memory abstraction on the Cell. Researchers are attempting to overcome these disadvantages by implementing smarter software caches. The work of Vujic et al. [95] augments a standard software cache so that it differentiates between high and low locality

accesses. Access patterns that the compiler recognizes will have high locality can benefit from prefetching, greatly increasing the likelihood of cache hits. Accesses that the compiler determines are unlikely to result in cache hits, and is unable to prefetch for, go into a separate cache with a lower miss overhead. The work of Chen et al. [25] attempts to improve the hit rate of software caches by prefetching irregular references—memory accesses that are not affine expressions of the loop induction variable. Inner loops are split into address collection loops and computation loops. Address collection loops execute first, and they only execute the code necessary to collect the addresses that the irregular accesses require. The runtime system can then prefetch those addresses so they will be more likely to be in the cache during the execution of the computation loop.

Such optimizations leverage the fact that the software cache can be completely controlled by the compiler, unlike hardware caches. But the SPEs in the Cell were purposefully designed to not have a hardware controlled cache so that they could have the potential for higher performance. Die area that would have been consumed by the hardware cache could go to the local store, register files and the interconnect bus. Implementing in software what was decided against in hardware means the software caches will start with a considerable performance disadvantage.

4.2 Applications

For evaluating Cellgen's dynamic buffer size choice and its model accuracy, we use two scientific applications that were hand-optimized for Cell using IBM's SDK 3.0. We deliberately chose real applications optimized for the Cell architecture so that we can have a solid performance baseline to compare against. The two applications we use are PBPI [35] and Fixedgrid [50, 71, 89].

PBPI is a parallel implementation of the Bayesian phylogenetic inference method [35], which constructs phylogenetic trees from DNA or AA sequences using a Markov chain Monte Carlo (MCMC) sampling method. On Cell, calculation of the likelihood values of each generation is distributed among SPEs.

For our experiments, we used a data set of 107 taxa with 9,994 nucleotides for a tree. There are three computational loops that are called for a total of 13,006 times and account for the majority of the execution time of the program. The first loop accounts for 84% of the calls, and requires 624 KB to compute a result of 312 KB; the second loop accounts for 8% of the calls and requires 936 KB to compute a result of 624 KB; and the third also accounts for 8% of the calls and requires 354 KB to compute a result of 8 bytes.

Fixedgrid is an atmospheric modeling application. It describes chemical transport via a third order upwind-biased advection discretization and second order diffusion discretization [71]. To calculate mass flux on a two-dimensional domain, a two-component wind vector, horizontal diffusion tensor, and concentrations for every species of interest must be calculated.

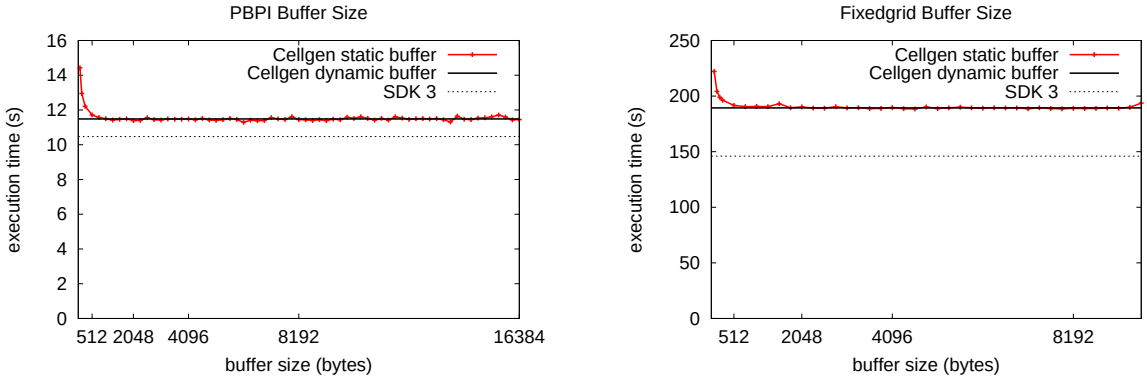


Figure 4.2: Buffer size evaluation for PBPI (left) and Fixedgrid (right). *Cellgen static buffer* represents performance when a static size is used, *Cellgen dynamic buffer* represents performance when Cellgen determines the best buffer size at runtime, and *SDK 3* represents the performance of the original, hand-optimized version of the application.

To calculate ozone concentrations on a 600×600 domain, approximately 1,080,000 double-precision values (8.24 MB) are calculated at each time step and 25,920,000 double precision values (24.7 MB) are used in the calculation. These calculations access non-contiguous data, which requires using DMA lists for best performance. Fixedgrid performs the discretization on both the rows and columns of the two-dimensional domain.

Both applications were modified to use non-vectorized kernels. We want to focus on the performance differences that arise from Cellgen's strategies for scheduling data transfers and those available to an expert programmer.

4.3 Dynamic Buffer Size Selection

To evaluate Cellgen's choice for dynamic buffer size (Equation 3.1 in Section 3.8), we compare the dynamic choice against static buffer sizes with a granularity of 256 bytes, as shown in Figure 4.2. We use versions of PBPI and Fixedgrid implemented with Cellgen; the static version uses the specified buffer size, and the dynamic version uses Equation 3.1. As a performance baseline, we also compare against the original, hand-optimized version of each application implemented with IBM's SDK 3.0.

The crossover point where computation and communication are overlapped happens relatively early for both PBPI and Fixedgrid—for PBPI this happens at 512 bytes, and at 768 bytes for Fixedgrid. For PBPI, Cellgen chooses buffer sizes of 16 KB, the maximum transfer size. The size of each loop is 9994 doubles, so from Equation 3.1, $N_b = 79952$ bytes. Since

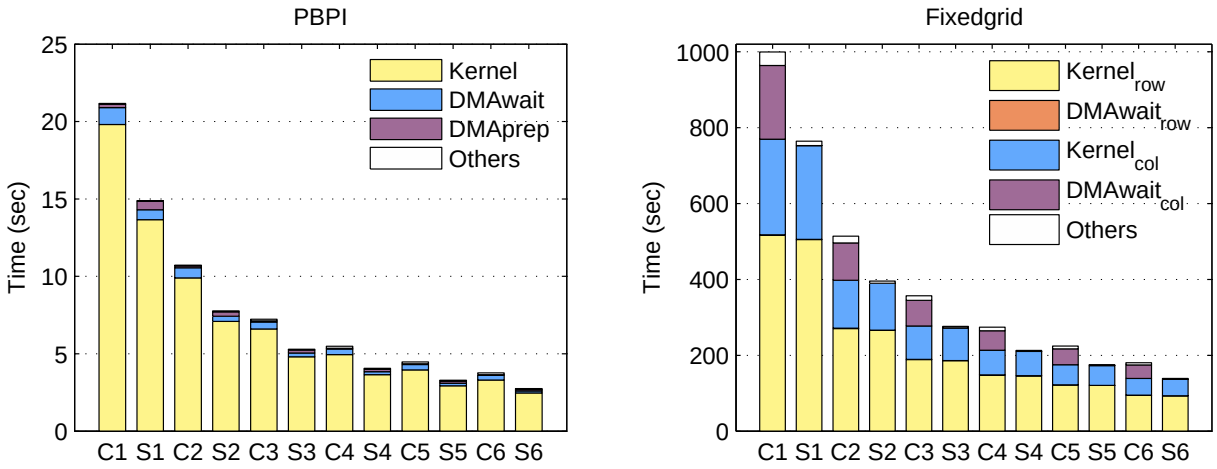


Figure 4.3: SPE timing breakdown for PBPI (left) and Fixedgrid (right). Each bar represents the average of 10 runs, and is labeled with its implementation (C for Cellgen and S for SDK 3.0) and the number of SPEs used. *Others* accounts for signaling overhead between the PPE and SPE, and all overheads related to issuing a DMA (except for PBPI where DMA preparation cost is listed separately).

the loop only has strict *in* or *out* variables, $d = 2$, which makes the calculated buffer size 26650 bytes. Since the ideal transfer size is greater than the maximum amount of data that can be transferred in a single DMA, Cellgen chooses the maximum DMA size of 16 KB.

The data set size for each Fixedgrid loop is 600 padded doubles. We pad each double to 16 bytes to achieve correct data alignment. Hence, each Fixedgrid inner loop requires transferring $N_b = 3200$ bytes. Cellgen classifies each variable as strict *in* or *out*, so $d = 2$ and the calculated buffer size is 3200 bytes. This size is less than the maximum transfer size, so it is used at runtime.

4.4 Application Performance

Figure 4.2 shows total application performance of the Cellgen implementation of PBPI comes within 9% of the hand-optimized version. Detailed timings from the SPE portions of the code are shown in Figure 4.3. This data shows Cellgen is able to generate code that can overlap communication and computation similarly to what an expert programmer can achieve, with communication overhead as low as 12% longer than the communication overhead of expertly tuned code.

As shown in Figure 4.3, both the Cellgen and hand-optimized version of Fixedgrid have negligible communications costs for the row discretization kernel—never more than 0.35% of the total SPE execution time. Cellgen's column discretization, however, has communication

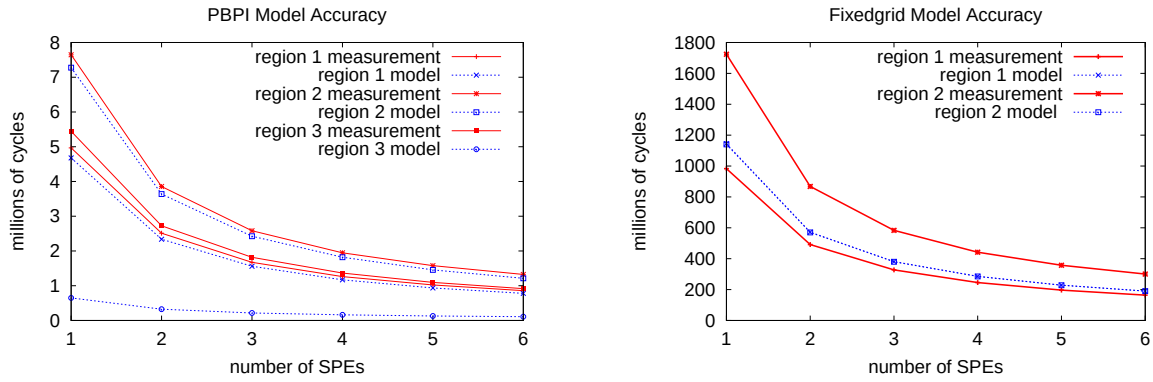


Figure 4.4: Model accuracy evaluation for PBPI (left) and Fixedgrid (right). We present the measured and predicted cost for each accelerated region.

time that is 48 times longer, which accounts for Cellgen's total application performance coming within 23% of the hand-optimized version. The memory latency in Cellgen's column discretization kernel is not overlapped with computation. The Cellgen version is written such that all of the data for a row or column is copied by the programmer into a local buffer. This buffer is different from the buffers Cellgen generates. Having the entire row or column is necessary because iterations of the computations are not strictly independent; there are loop carried dependencies spanning 3 consecutive loop iterations. Cellgen is not able to natively handle such circumstances in the current implementation. When computing rows, the amount of time spent copying the data is not large compared to the computation. When computing columns, the amount of time spent copying the data is significant—it accounts for 20% of SPE execution time.

4.5 Model Accuracy

We present the results from the experiments comparing Cellgen's model prediction against the actual measurements for each accelerated region in Figure 4.4.

For all three accelerated regions in PBPI, Cellgen correctly predicts that the regions will be computation dominated as opposed to communication dominated. Cellgen further predicts correctly that actual computation on pipeline 1, as opposed to loads and stores on pipeline 0, will dominate. For region 1, Cellgen is off by 5.8% with one SPE, and 9.2% for all six SPEs. For region 2, Cellgen is off by 4.8% with one SPE, and 8.5% for all six SPEs. For these regions, Cellgen under-predicts because the cycle cost estimate only considers the main computation involved, and ignores the cost of starting up the computation.

For region 3, Cellgen is off by a factor of 8.3 for one SPE and 8.4 for all six SPEs. The problem with region 3 is that the computation is dominated by a call to a shared library logarithm function. Cellgen only recognizes the computations presented in Table 3.1. While, in principle, the cost of a logarithm function will depend on those operations, the function call itself is a black box. This misprediction highlights one of the limitations of the approach we have taken to performance prediction with Cellgen: we assume the actual computations that dominate the cost of an SPE region will appear in the accelerated region. When this assumption does not hold, as in region 3, Cellgen's model is not accurate.

The two accelerated regions in Fixedgrid perform the same computations; they differ only in how they access data. Hence, the model predictions for both region 1 and region 2 for Fixedgrid are the same. Cellgen's prediction for region 1, which performs row accesses, maintains a misprediction error of 14% for all number of SPEs. However, for region 2, the misprediction error is between 34–36%. The problem with region 2 is that it accesses memory in columns, and the model for DMA transfers (Equations 3.3 and 3.4 in Section 3.9) assumes single DMA transfers. Transferring column data is more expensive because Cellgen must generate the DMA lists themselves, and transferring N bytes of data in a single DMA is going to be less expensive than transferring N bytes of data split up in many different chunks, as is done with DMA lists. The solution is that we need to generalize our model to include strided access to memory that requires the generation of DMA lists. The generalization of the model should include the cost to generate and to schedule DMA lists. Cellgen already knows at compile time if a transfer will require DMA lists, so we can easily extend the compiler itself to handle the extended model.

4.6 Stencil Benchmarks

In order to test the efficiency of the stencil access techniques presented in Section 3.7, we present experimental results that test these techniques against naive code generation and an expertly-tuned application.

4.6.1 Transfer Savings

Figure 4.5 presents the results of an experiment which compares the performance of the code generated for stencil accesses (labeled *stencil* in the figures) against the performance of performing a DMA transfer for each separate access (labeled *naive* in the figures). There are two implementations: one which sums up the cardinal neighbors of each element in a matrix (labeled with 4 in the figures) and one which sums up the cardinal and ordinal neighbors of each location (labeled with 8 in the figures). We use square matrices of sizes 64×64, 128×128, 1024×1024 and 2048×2048 integers. Matrices of size 64×64 are at the edge of where it is beneficial to offload the computation to the SPEs, and matrices of size

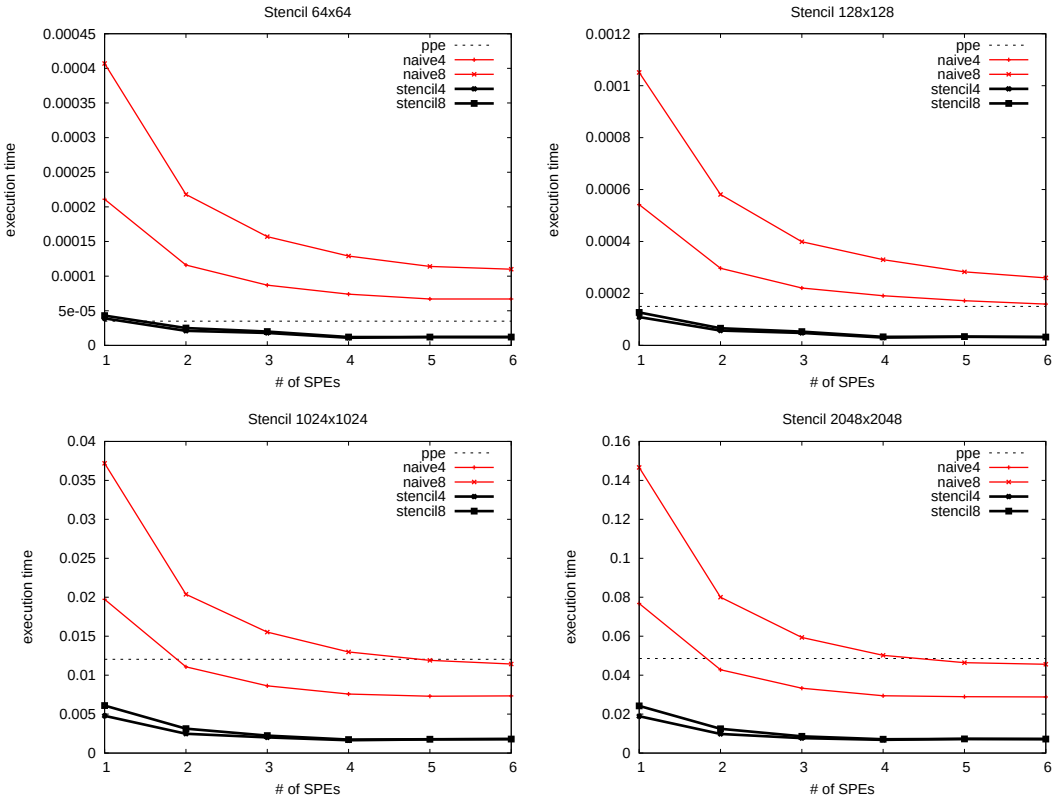


Figure 4.5: Benchmarks comparing the performance of Cellgen's rotating system of buffers (*stencil*) with code that performs a transfer for every access (*naive.*). There are two variants: accessing every cardinal neighbor (labeled as 4) and accessing every cardinal and ordinal neighbor (labeled as 8). Each graph represents experiments with a square matrix whose dimension is in the title. The line labeled *ppe* in each graph represents how long it took to perform the computation on the PPE.

2048×2048 are at the edge of the amount of data than effectively be kept in a PS3's main memory.

The experiments in Figure 4.5 show the time saved by using the rotating system of buffers versus using separate transfers for each stencil access. Using all six available SPEs, the system of rotating buffers performs up to 2 times faster when accessing the cardinal neighbors and up to 3 times faster when accessing the cardinal and ordinal neighbors. Cellgen's stencil code also performs nearly the same with both the cardinal-only and cardinal-ordinal versions. This performance similarity exists because both versions execute the same number of transfers, just with one extra element at the beginning and one extra element at the end of each buffer.

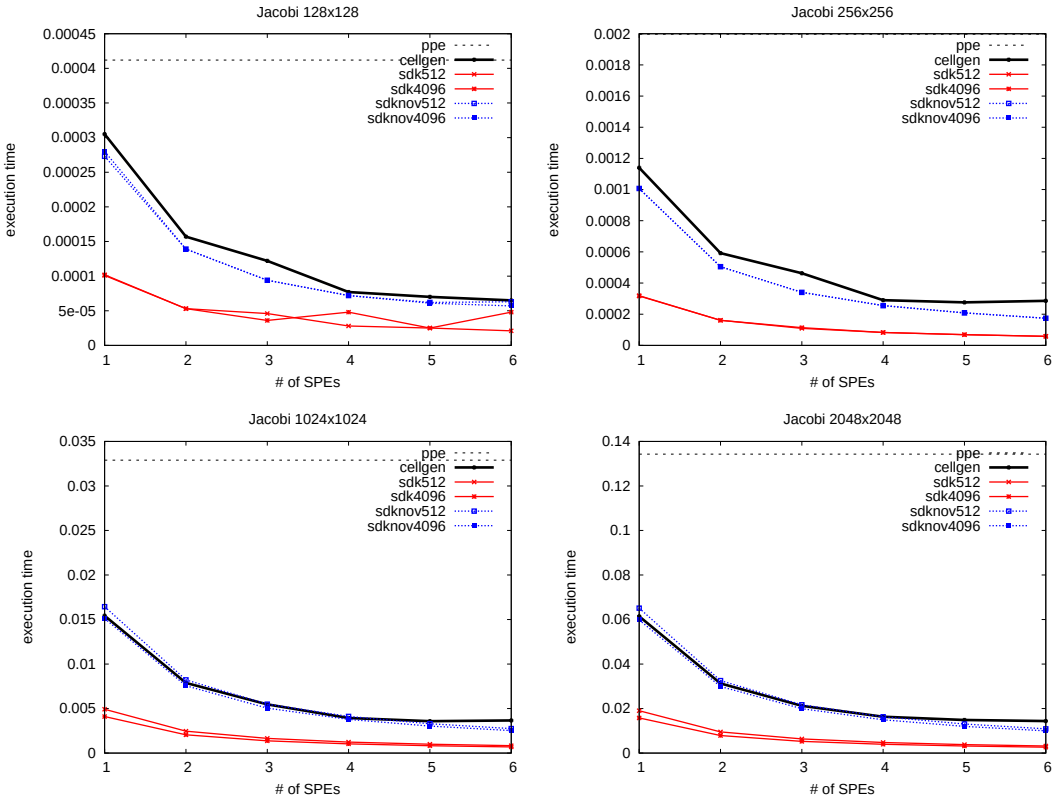


Figure 4.6: Performance of Cellgen and SDK implementations of the Jacobi computation. The SDK implementation has both vector versions (labeled *sdk*) and non-vectorized versions (labeled *sdknov*). The SDK versions are also varied by the buffer size used in each transfer, 512 bytes or 4096 bytes. The line *ppe* represents the time taken to perform the computation on the PPE.

4.6.2 Jacobi

We compare a Cellgen version of the Jacobi computation to an expertly-tuned version implemented with the Cell SDK. Both implementations compute the average of four floating point numbers in the cardinal directions to each element in a matrix. We present results for the SDK implementation with and without vectorization (labeled *sdk* and *sdknov* respectively). Cellgen does not perform automatic vectorization and we want to focus on the data transfer strategies. The SDK implementation also takes the amount of data to be in each DMA transfer as a parameter; our experiments used 512 bytes and 4096 bytes.

Vectorization makes a significant performance difference for the SDK version, and the difference increases along with the size of the matrix. For the largest matrix, the vectorized code is almost 4 times faster than the unvectorized code—which is expected, since the SPEs can operate on 4 floats in a single vector instruction. The smaller matrix sizes benefit from vectorization, but speedups around a factor of 2 reflect that the ratio of vectorized instructions to other overheads decreases. Varying the buffer size in the SDK version from 512 to 4096

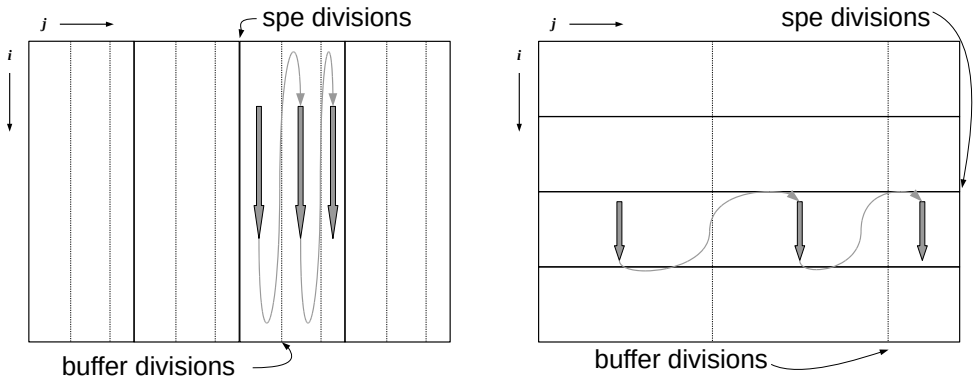


Figure 4.7: Matrix division strategies for Cellgen (left) and the SDK implementation of the Jacobi computation (right). The label *spe divisions* indicates how the data is divided among the SPEs (vertically for Cellgen and horizontally for the SDK version). The label *buffer divisions* indicates how the data is split into buffers for each SPE (vertically for both implementations).

improves performance by about 20%, which implies that larger DMA transfers are better.

We want to focus on the data transfer strategies when comparing the performance of the SDK version to the Cellgen implementation. Hence, we will focus on how the Cellgen version compares to the non-vectorized SDK versions.

There are two major differences between Cellgen's data transfers and the SDK version. The first difference is that each floating point number in the Cellgen implementation is padded to 16 bytes. Cellgen requires this padding so that it can perform arbitrary data transfers among all elements. The second difference is presented in Figure 4.7, which are the different data division strategies. Cellgen's strategy is to first divide the matrix vertically among all of the SPEs, and then further divide the vertical SPE slices into buffer-sized slices as needed. In contrast, the SDK version first divides the matrix horizontally among the SPEs, and then divides the horizontal slices vertically into buffer-sized slices. As the size of the matrices increases, these two strategies will converge in terms of total number of DMAs. However, for the sizes used in our experiments, the strategy used by the SDK implementation is better: it performs up to 10 times fewer total DMAs. The reason for this discrepancy is that as more SPEs are used, Cellgen's strategy does not decrease the number of DMAs required by each SPE. If the matrix is $N \times N$, each SPE will have at least N DMA transfers (although the total data actually transferred will change). The SDK implementation's strategy decreases the number of DMAs as SPEs are added. This behavior explains why the performance for Cellgen flattens out at higher than 4 SPEs, while the SDK implementation sees continued improvement. In the future, we plan to integrate the strategy the SDK implementation uses into Cellgen.

The final performance anomaly with Cellgen's Jacobi results is the "bump" seen at 3 SPEs for matrices of 128×128 and 256×256. The cause of this bump is due to the fact that when 2 and 3 SPEs are used, the data division results in a remainder; there were many DMAs of

small amounts of data. For 4 SPEs and up, the data division is even; it has no remainder.

4.7 Summary

While programming with Cellgen is similar to programming with OpenMP, it is not exactly the same for key reasons. Our approach to supporting a shared memory abstraction is to only accept memory access patterns that we can generate high performance code for. Not supporting all access patterns is in contrast to the software cache approach. Cores in a processor are typically connected to main memory through a hardware controlled cache. The vector cores on the Cell are not connected to main memory in the conventional way; there is no hardware controlled cache, only DMAs. The software cache approach to supporting a shared memory abstraction replaces all accesses to shared data with calls into a software cache that exists on all vector cores. While this approach does support all memory access patterns, it does so to the detriment of performance, as we demonstrated in Section 4.1. These results should generalize to any heterogeneous multicore architecture where at least some of the cores are not a part of the normal memory hierarchy.

Chapter 5

Programming Models: An Evaluation of Different Programming Abstractions for Heterogeneous Multicores with Explicitly Managed Memories

Cellgen represents an implicit programming model for data parallelism. In this chapter we explore other means for programming processors with explicitly managed memory hierarchies, both qualitatively and quantitatively. Specifically, we compare Cellgen with Sequoia [34] and the Cell SDK [52]. Sequoia's goal is to abstract the memory hierarchy into levels, and programmer's explicitly state the data division and computation that happens at each level. The Cell SDK is a thin wrapper over the features provided by the hardware. Table 5.1 shows the design space we explore.

Table 5.1: Programming model exploration space.

Programming model	Task creation	Granularity	Locality	Scheduling data transfers
Cellgen	implicit	implicit	implicit	implicit
Sequoia	explicit	explicit	explicit	implicit
Cell SDK 3.0	explicit	explicit	explicit	explicit

5.1 Cellgen

In prior chapters, we explained how to program with Cellgen, and how the compiler transforms high-level code into Cell code. We build on this presentation by presenting Cellgen

code taken from PBPI [35]:

```
int N4 = N/4;
#pragma cell private(int N4 = N4, double* freq = model->daStateFreqs) \
      shared(double* sroot = tree->root->siteLike, int* weight = g_ds.compressedWeight.v) \
      reduction(+: double l = lnL)
{
  int i;
  for (i = 0; i < N4; i++) {
    double temp;
    temp = sroot[(i*4)]*freq[0] + sroot[(i*4)+1]*freq[1] +
           sroot[(i*4)+2]*freq[2] + sroot[(i*4)+3]*freq[3];
    temp = log(temp);
    l += weight[i] * temp;
  }
}
```

This code is part of the maximum likelihood calculation. It shows several Cellgen features used in a real application—shared variables, private variables, implicit reference analysis and reductions. We will compare this code to the Sequoia version.

5.2 Sequoia

The second class of programming models that we consider in this study expresses parallelism through explicit task and data subdivision. We use Sequoia [34] as a representative of these models. In Sequoia, the programmer constructs trees of dependent tasks where the inner tasks call tasks further down the tree; the real computation typically occurs in the leaves. At each level, the data is decomposed and copied to the child tasks as specified, which enforces the model that each task has a private address space. The following Sequoia code performs the same computation as the Cellgen code from the previous section:

```
void task<inner> Likelihood::Inner(in double sroot[N],
  in double freq[M], in int weight[P], out double lnL[L])
{
  tunable T;
  mapreduce (unsigned int i = 0 : (N+T-1)/T) {
    Likelihood(sroot[i*T;T], freq[0;3], weight[i*T/4;T/4], reducearg<lnL, Sum>);
  }
}
```

```
void task<leaf> Sum::Leaf(in double A[L], inout double B[L])
{
  B[0] += A[0];
}

void task<leaf> Likelihood::Leaf(in double sroot[N],
  in double freq[M], in int weight[P], inout double lnL[L])
{
  unsigned int i;
  for (i = 0; i < P; i++) {
    unsigned int j = 4 * i;
    double temp = sroot[j] * freq[0] + sroot[j+1] * freq[1] +
                  sroot[j+2] * freq[2] + sroot[j+3] * freq[3];
    temp = log(temp);
    lnL[0] += weight[i] * temp;
  }
}
```

Sequoia uses notation similar to C++ classes to represent computations. The appropriately named member functions of that class are how the programmer describes what to do at each level of the hierarchy. The computation itself is contained in the leaves, which in this example is in Sum::Leaf and Likelihood::Leaf. The function Sum::Leaf specifies how to perform the summation reduction. The function Likelihood::Leaf performs the actual computation on the subset of the data passed into it. Both functions are tagged with task<leaf> to indicate to the compiler that these are leaf tasks. All values passed into functions are also explicitly labeled with **in**, **out** or **inout**.

The data division occurs in the function Likelihood::Inner, which is tagged as an inner node in the tree with task<inner>. This division is expressed in Sequoia with the **mapreduce** iteration construct. The implication from the **map** prefix is that the Likelihood computation will be applied to all data subdivisions—**mappar** exists for computations that can occur in parallel but have no reduction, and **mapseq** is for computations that must be sequential. The data is divided in terms of the induction variable, i, and the **tunable** value T. In Sequoia, **tunable** variables are used to generically express how data should be partitioned at each level. Sequoia uses a hardware specification file to generate a suitable constant at compile time. Finally, the **reducearg** construct is used to indicate that the variable lnL should be reduced using the computation Sum.

Locality is strictly enforced by Sequoia because tasks can only reference local data. In this manner, there can be a direct mapping of tasks to the Cell architecture where the SPE local storage is divorced from the typical memory hierarchy. By providing a programming model where tasks operate on local data, and providing abstractions to subdivide data and pass it on to subtasks, Sequoia is able to abstract away the underlying architecture from programmers. Sequoia allows programmers to explicitly define data and computation subdivision through

a specialized notation. Using these definitions, the Sequoia compiler generates code which divide and transfer the data between tasks and performs the computations on the data as described by programmers for the specific architecture. The mappings of data to tasks and tasks to hardware are fixed at compile time.

In Sequoia, application users may suggest to the compiler certain optimization approaches such as double buffering for transferring data and an alternative strategy of mapping data divisions to subtasks. The compiler generates optimized code based on these hints, if possible.

Sequoia tries to ensure that programmers are free from the awareness of the architectural constraints such as the DMA size and data alignment requirements by providing an interface to allocate arrays. Programmers are expected to use such an interface to allocate arrays which are handled by Sequoia. When a programmer makes a request for an array of a particular size, the amount actually allocated by Sequoia may be larger. It must be at least 16 bytes to satisfy the DMA size constraint, and it must be a multiple of 16 bytes to satisfy the DMA alignment constraint. Additionally, Sequoia must allocate memory for the data structure that describes the array.

Sequoia also provides an interface to copy an ordinary array to an array allocated by the Sequoia interface. When applying the Sequoia framework to a given reference code, programmers allocate Sequoia arrays and copy existing arrays to the Sequoia counterparts before calling the Sequoia computation kernel. Conforming to this assures that the architectural constrains are satisfied at the cost of additional copying overhead.

In our experiments, we use the customized array allocation interface which allocates space for the Sequoia array structure only—it does not actually allocate any extra data. We then manually point the Sequoia array data structure to the existing data. Therefore, the DMA constraints are explicitly taken care of in our experiments when an array is allocated. This technique avoids unnecessary copies from application managed data to Sequoia managed data, as well as unnecessary allocations.

5.3 Cell SDK

Our third programming model is the Cell SDK 3.0, as provided by IBM. The SDK exposes architectural details of the Cell to the programmer, such as SIMD intrinsics for SPE code. It also provides libraries for low-level, Pthread style thread-based parallelization, and sets of DMA commands based on a get/put interface for managing locality and data transfers.

Programming in the Cell SDK is analogous, if not harder, than programming with MPI or POSIX threads on a typical cluster or multiprocessor. The programmer needs both a deep understanding of thread-level parallelization and a deep understanding of the Cell hardware.

While programming models can transparently manage data transfers, the Cell SDK requires all data transfers to be explicitly identified and scheduled by the programmer. Furthermore,

the programmer is solely responsible for data alignment, for setting up and sizing buffers to achieve computation/communication overlap, and for synchronizing threads running on different cores. However, hand-tuned parallelization also has well-known advantages. A programmer with insight into the parallel algorithm and the Cell architecture can maximize locality, eliminate unnecessary data transfers and schedule data and computation on cores in an optimal manner.

5.4 Qualitative Comparison

The most obvious difference between the Cellgen and Sequoia code samples is length. What Cellgen can express in 13 lines, Sequoia requires 26. This brevity comes from the fact that Cellgen is designed for data-parallel code, but it is instructive to examine exactly what is implicit and explicit between Sequoia and Cellgen.

Data Direction Cellgen uses reference analysis to infer from the programmer's code if variables are *in*, *out* or *inout*. Sequoia requires the programmer to explicitly state the data direction for each variable at all levels—note that in the Sequoia code, the programmer must duplicate effort by redeclaring all of the variables passed into Likelihood::Inner for Likelihood::Leaf. Duplication of effort implies that the introduction of bugs is more likely when programmers change code [83].

Data Division The Sequoia code uses the **mapreduce** construct and a **tunable** variable to describe how the data should be divided. Cellgen infers how the data should be divided among the SPEs based on how the programmer uses the variables.

Locality Sequoia controls locality through **tunable** parameters. While the compiler replaces this parameter with a static value at compile time, that value comes from a configuration file specific to the target architecture. Cellgen determines locality through both static and runtime information, as explain in Section 3.8.

Both programming models require some construct to say "this is a reduction." However, the Sequoia construct requires the reduction to be placed in a separate leaf function, while the Cellgen construct allows the programmer to state the reduction inline with the rest of the code.

Requiring more language constructs to implement the same computation is not necessarily a weakness for Sequoia. While there is marginal benefit in the case of data parallel code, Sequoia's explicit constructs allow it to be used in more general ways. It is a more general means to express parallelism, and can be applied to programming paradigms that break Cellgen's assumptions.

Both Cellgen and Sequoia are clear programmability wins over coding directly with the Cell SDK. We have focused most of our discussion here on how parallelism is expressed on a particular computation. This ignores one of the major benefits that any programming language for EMM processors will provide: infrastructure. By using a programming language that targets a particular architecture, programmers are freed from having to deal with details unrelated to their application. We have focused most of our communication discussion on DMAs because they have a large impact on performance, but there is more PPE-SPE communication that must occur just to initiate a computation on SPEs. This convenience is analogous to the ease with which C programmers can define functions and ignore assembly-level details such as how the parameters are accessible from within the function, or how the function knows where to resume execution upon completion.

5.5 Quantitative Comparison

For a quantitative analysis, we compare the performance of each implementation of each application. The experimental environment is a Sony PlayStation 3 running Linux with a 2.6.24 kernel and Cell IBM SDK 3.0. On a PS3 running Linux, only six SPEs are available. The compiler is gcc 4.1.2 with full optimizations turned on. Each data point, except for CellStream, represents the best of 40 runs; we found this more reproducible and representative than the average. For CellStream, the average is more appropriate.

5.5.1 CellStream

CellStream [88] is a benchmark designed to transfer a large amount of data from main memory through the SPEs as fast as possible. The goal is to come as close as possible to achieving the theoretical bandwidth peak of the Memory Interface Controller. Figure 5.1 shows the bandwidth achieved by CellStream implemented with two different version of Cellgen, two different versions with Sequoia, the original version implemented with SDK 3.0, and a version which uses only the PPE. The test streamed a 192 MB file cached in memory through a single SPE in 16 KB chunks (except in the PPE only version). The bandwidth was calculated by dividing the total data transferred by the amount of time spent working. Any extra time spent by the PPE threads reading in and writing to disk was not factored in. The PPE only version ran solely on the PPE and modified the memory by using a memset call on the data. It serves as a baseline for comparison, as it represents what can be achieved without the aid of the SPEs.

Figure 5.1 measures the bandwidth of streaming the data from main memory to the SPE and back to main memory. Data streaming requires a DMA get, some processing, and a DMA put command. The bandwidth calculations in Figure 5.1 are the total data transferred divided by the time the SPE was busy. This SPE busy time includes reading and writing of

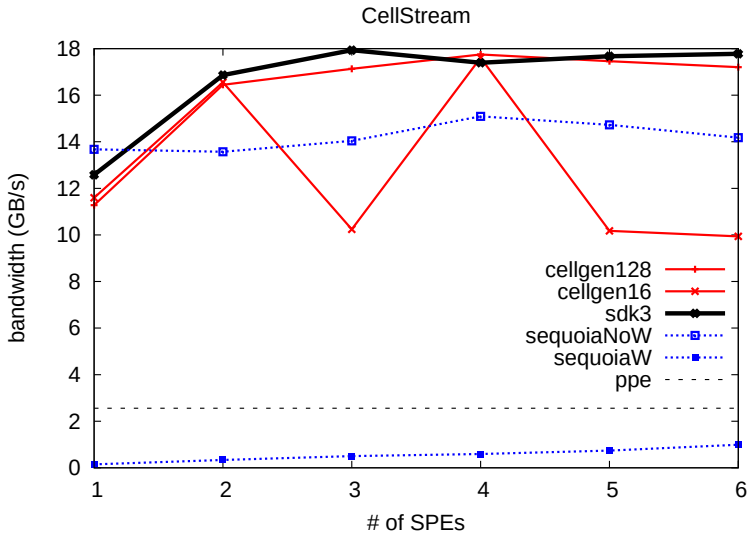


Figure 5.1: Average bandwidth achieved by each implementation of CellStream as a function of the number of SPEs used.

the data, so each 16 KB of data had two DMA operations performed on it.

The two different versions of Cellgen used with CellStream have different scheduling policies. Specifically, one parameter was changed in compute_bounds, which is explained in detail in Section 3.6. The approach described there tries to find the most even division of the iteration space. We accomplish this by dividing the space in 16-byte chunks. While this approach does achieve a fair division of work, it does not respect the cache line size of the architecture—it is possible to have data divisions that are not 128-byte aligned, which is the size of a cache line. For 1, 2 and 4 SPEs, the data transfers happen to be aligned on cache lines. Since CellStream is, by design, communication dominated, the cost of transferring data not aligned on a cache line matters more than having an equal division of work among the SPEs. The alternative version of Cellgen divides the iteration space in 128-byte chunks, which respects cache line sizes at the cost of a less even division of work among the SPEs. In the future, we will use the prediction presented in Section 3.9 to predict if an accelerated region is computation or communication dominated. For computation dominated regions, a scheduling policy which favors an even distribution of work is best. For communication dominated regions, a scheduling policy which respects cache line sizes is best.

We implemented CellStream in two different ways with Sequoia. The first, labeled sequoiaW in Figure 5.1, does the same work as the Cellgen and SDK3 versions on the SPE: set each byte to the character '1'. With both Cellgen and the SDK3 versions, this operation is vectorized. (Cellgen is not a vectorizing compiler, but it will preserve code that has already been

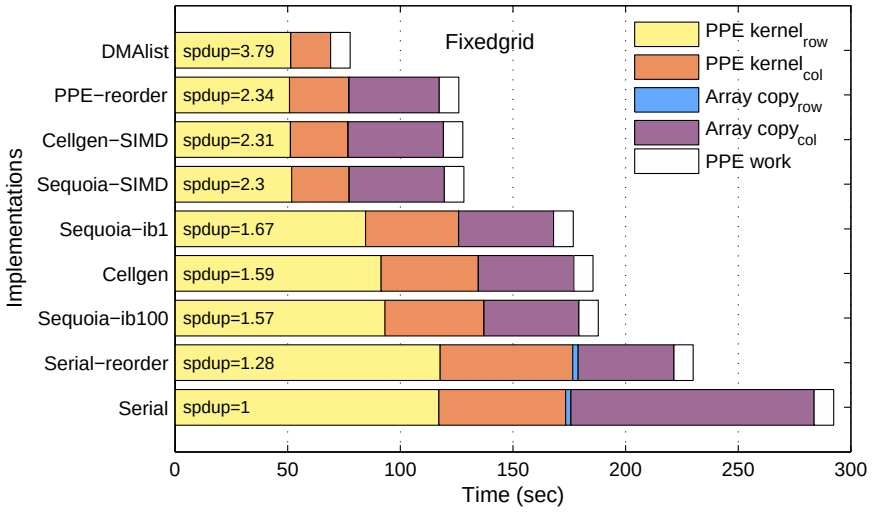


Figure 5.2: Whole-application timing profile for Fixedgrid. *PPE kernel* measures the time to complete each offloaded function, from the perspective of the PPE. *Array copy* is the time spent on copying arrays from main memory for each discretization function. *PPE work* includes array initialization and file I/O time.

vectorized. Sequoia will not.) The Sequoia version labeled sequoiaW uses an unvectorized for loop to set its buffer elements. The code that Sequoia generates for the SPE has three pointer dereferences for every buffer access. Consequently, the Sequoia code must perform 16 times the amount of stores due to not being vectorized, and each of those stores requires three additional loads to determine the correct local store address. This behavior makes the Sequoia version computation dominated. In order to evaluate Sequoia's communication scheme, sequoiaNoW performs no work; the SPEs do communication only. It represents the bandwidth Sequoia can achieve after further SPE code optimization.

All three implementations issue 24,576 DMA calls of 16 KB each (12,288 reads and 12,288 writes). The only difference between the hand written version and the models is the use of double buffering with DMA fencing calls in the former and triple buffering in the latter. The fencing calls used in the hand written version have shown a slight speed up compared to using an extra buffer.

The main bottleneck in all implementations is the Memory Interface Controller. The MIC can access the memory at a rate of 16 bytes per bus cycle. The EIB operates at 1.6 GHz in our experiments (half of the CPU clock frequency) which gives a theoretical peak speed of 25.6 GB/s.

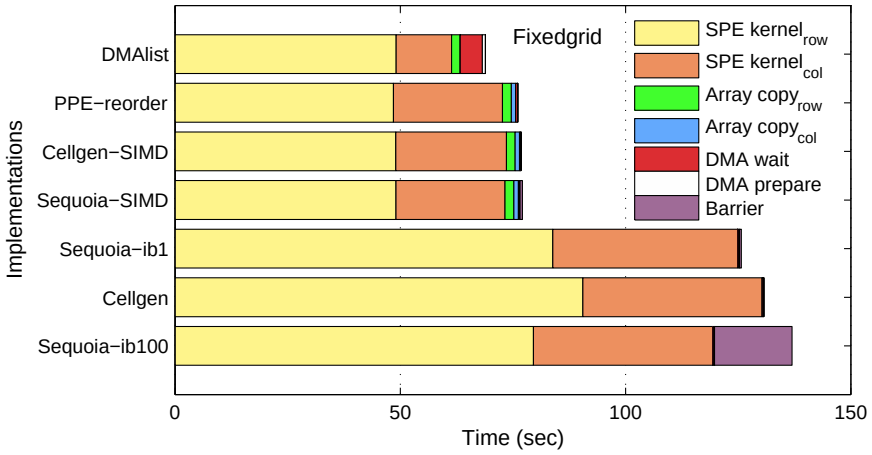


Figure 5.3: SPE-kernel timing profile for Fixedgrid, which shows the time breakdown of offloaded functions. *SPE kernel* is the time spent on core computation excluding DMA and array copying overheads. *Array copy* is the time spent on copying arrays to SPE local storage. *DMA wait* is the DMA data transfer time not overlapped with computation in addition to the time for checking the completion of DMA commands. *DMA prepare* is the time to prepare DMA addresses and lists in addition to the time to queue DMA commands.

5.5.2 Fixedgrid

We experimented with nine different implementations of Fixedgrid to see how the handling of non-contiguous data transfers affects the performance of the application. These implementations represent the choices for how an implicit model can abstract strided accesses. The version of Cellgen we used could not handle handle strided access to memory. Consequently, both the Cellgen and Sequoia versions of Fixedgrid only access contiguous memory from the SPEs.

Fixedgrid has two types of computational kernels: the row discretization kernel and the column discretization kernel. The former requires row data from a contiguous region of memory, and the latter requires column data from a non-contiguous region of memory. For each time-step iteration, the former is called twice as much as the latter is. In the serial version, column data is copied to a contiguous buffer as each column is needed by the column discretization kernel running on the PPE. The row/column discretization kernel requires a row/column from three different matrices to compute a result row/column.

The serial-reorder version maintains a transposed copy of each matrix. Therefore, no buffer is used in the column discretization kernel in contrast to the serial version. Instead, the values of each transposed matrix are copied as a whole from the original matrix before the column discretization kernel. They are then copied back as a whole to the original matrix after the computation. The kernel accesses data directly from the transposed matrix. This version benefits from higher locality than the serial version since the transformation from

row-major to column-major format is grouped by each matrix. Consequently, the serial-reorder version spends less time copying column data as shown in Figure 5.2. The rest of the implementations—except DMAlist—are based on the serial-reorder version. Therefore, they all spend similar amount of time copying column data on the PPE. There is also a smaller amount of time spent copying row data in both of the serial versions. This operation is replaced by DMA calls in other versions.

The Cellgen and Sequoia versions are implemented with two types of SPE kernels: conventional operations (as in the serial versions) and SIMD operations. Both of the handed-coded versions, DMAlist and PPE-reorder, are only implemented with the SIMD kernel. The Cellgen and Sequoia versions with non-SIMD kernel show similar performance.

In Sequoia, we test two strategies for mapping data sub-blocks to subtasks, labeled Sequoia-ib100 and Sequoia-ib1. The difference between Sequoia-ib100 and Sequoia-ib1 is the mapping of data sub-blocks to subtasks. Mapping configuration files allow users to specify the interblock mapping strategy that Sequoia uses to decide how data is distributed to subtasks. When the interblock option is set to 100 (Sequoia-ib100), Sequoia performs a block distribution of data with a block size of 100; task_0 takes from block_0 to block_{99}, task_1 takes from block_{100} to block_{199}, etc. When the interblock option is set to 1 (Sequoia-ib1), Sequoia performs an interleaved assignment of data to tasks with a block size of 1; task_0 takes block_0, block_6, block_{12}, task_1 takes block_1, block_7, block_{13}, etc., given that there are six subtasks for six SPEs.

The Cellgen version assigns blocks of contiguous iterations to each SPE. The block size is determined at runtime so that each SPE gets as close to an equal amount of work as possible. This division of work is similar to the Sequoia version when the interblock mapping strategy is 100. It exhibits load imbalance due to the data dependency in computation cost, as shown in Figure 5.3.

The DMAlist implementation of Fixedgrid uses DMA lists to transfer columns of data. DMA lists are the only mechanism provided by the Cell to perform scatter/gather operations. Column accesses are achieved by constructing lists of DMAs for each element in the column. However, since the minimum size of a DMA is 16 bytes, and each element is an 8 byte floating point value, DMA lists transfer unnecessary data. The SIMD operations also work on the data that was transferred as an artifact of the minimum DMA size. Unlike the other versions, DMAlist does not require column data to be reordered on the PPE or SPE.

In the PPE-reorder version, to make the same kernel work on non-interleaved row data, it is obtained by a DMA transfer and is interleaved into a vector array twice as large as the data itself. This copy operation on SPUs introduces the row array copy overhead shown in Figure 5.3. Since columns also require copying—they are reorganized into contiguous arrays—both row and column discretization kernels rely on copying operations.

The computation of an array element in the Fixedgrid kernel is dependent on the preceding elements in the array. Therefore, it is not possible to further utilize vector operations by

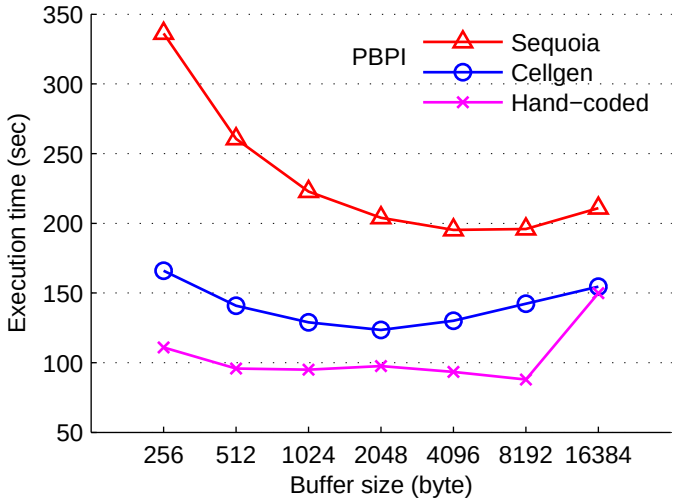


Figure 5.4: Total execution time as a function of DMA buffer size.

condensing a vector array. However, it is possible to utilize wasted vector operation cycles if two row/column elements are fetched and computed at the same time. When a row result is ready in an interleaved array, it is copied back to a non-interleaved buffer array on the local storage for bulk DMA transfer. Cellgen and Sequoia versions with the SIMD kernel rely on the same data rearrangement strategy as that of the PPE-reorder version. Overall, we find that the lack of support for automatic generation of DMA scatter/gather operations is the key reason for the performance gap between the high-level programming models and the hand-tuned version of Fixedgrid.

5.5.3 PBPI

Applications with a fine granularity of parallelism are sensitive to the size and frequency of DMAs between the SPE and main memory. Since PBPI is such an application, we experimented with different buffer sizes, as shown in Figure 5.4.

With the manual implementation, the optimal performance of PBPI was achieved with a buffer size of 8 KB. The best Sequoia performance was with a buffer size of 4 KB. With Cellgen, the best performance was achieved with a buffer size of 2 KB which is used in 64 unrolled iterations of a computational loop.

The principle behind loop unrolling on the SPE is to maximize the overlap of computation and communication. As the unroll factor increases, so does the amount of data transferred for each DMA. If the size of the DMA is too small, the data transfers can not keep up with the computation. But, as the unroll factor increases, so too does the code size, and eventually the code size becomes too large for the SPE. The best unrolling factor balances DMA size, computation time and code size. Cellgen programmers control loop unrolling by explicitly

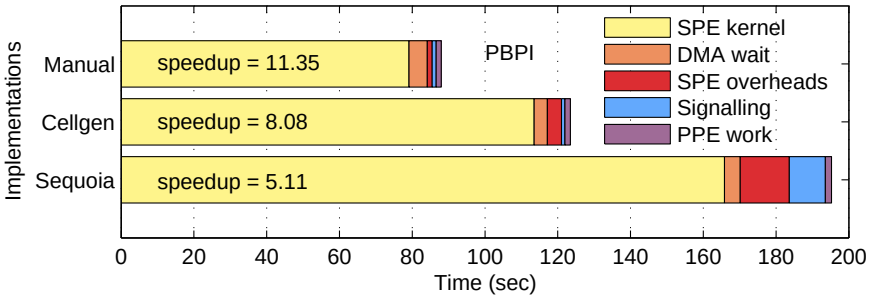


Figure 5.5: Comparison of the best cases from each implementation. *SPE kernel* accounts for the time to run hand-coded or generated SPE kernel. *DMA wait* is the DMA data transfer cost that is not overlapped with computation in addition to the time for checking the completion of DMA commands. *SPE overheads* account for DMA preparation, barriers, and other programming model specific overheads which vary depending on their implementations. *Signaling* accounts for overhead from signaling between PPE and SPE.

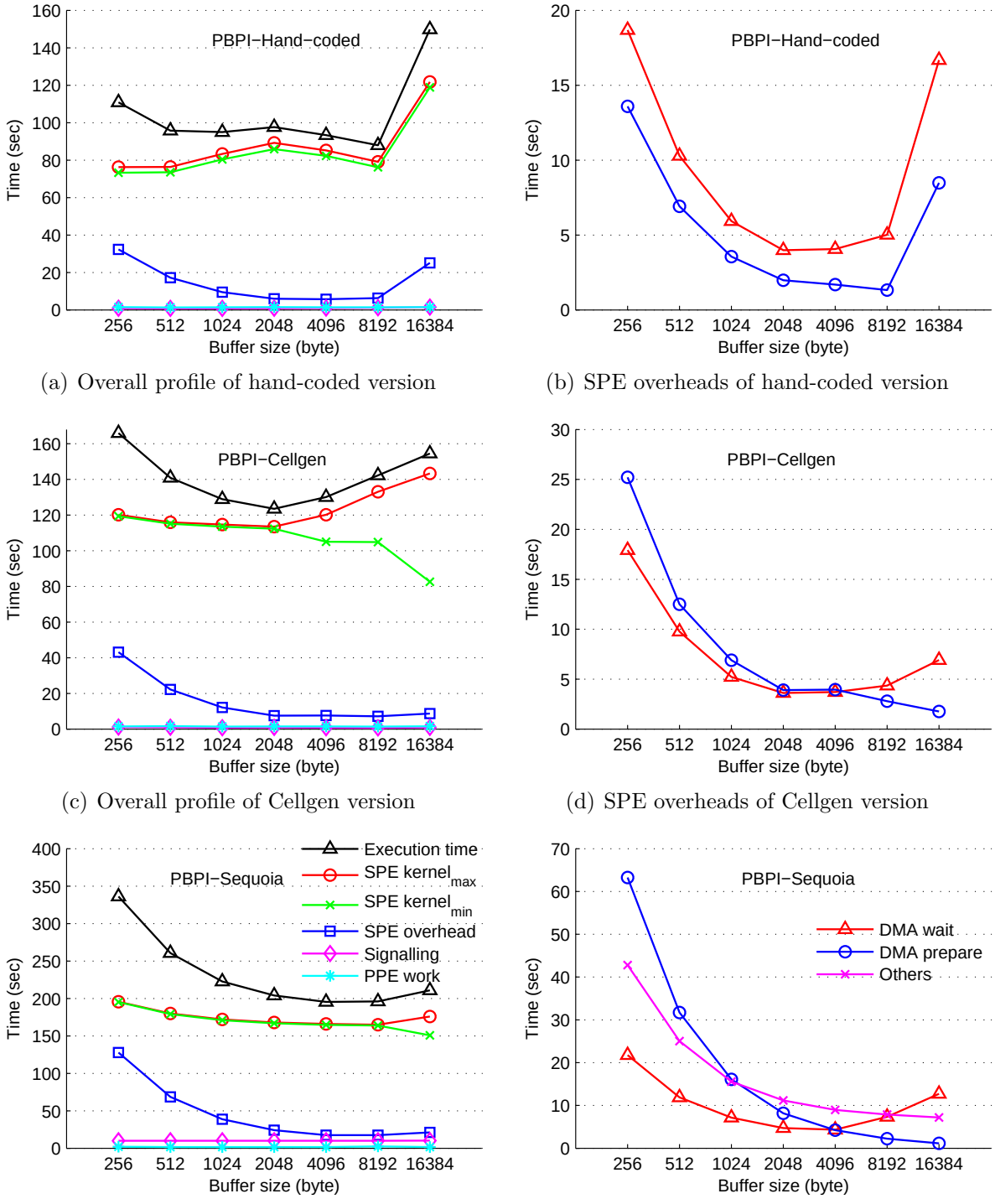
setting the unroll factor in the directive. Cellgen uses this unroll factor to choose a buffer size based on the use of the array inside the loop. These experiments are the groundwork towards deducing the best unroll factor at compile time.

The major factors that influence performance in all three cases are the performance of the computational kernel which is either manually written or generated for the SPE; the overhead of DMA related operations; the extra overheads on SPEs generated by the programming model runtime; and the overhead of signaling between PPE and SPE, as shown in Figure 5.5.

The SPE computational kernel generated by Sequoia relies on data structures to describe the array organization. Array accesses incur overhead due to the additional computation needed to translate the programmer's intent to Sequoia's data layout. This overhead is an instance of a programming model abstraction impacting performance. Similar overheads specific to Sequoia include the constraint checking for the size and alignment of DMA data and the DMA buffer padding to satisfy the constraints.

There are two differences between the SPE computational kernel generated by Cellgen and the computational kernel from the reference code: loop unrolling and a modulus operation introduced to each array access to translate a main memory address to an SPE buffer address. The hand-coded kernel is loop-unrolled and vectorized. The total execution time and the SPE kernel time of each implementation are shown in Figures 5.6(a), 5.6(c), and 5.6(e).

In Cellgen, the iterations are distributed to the SPEs once, before the computation starts, as opposed to dynamically on demand as the SPEs complete iterations. If the distribution of iterations is imbalanced, there may be variance in the time it takes for a single SPE to complete its iterations. The SPE that takes the longest holds up the rest of the computation. The imbalance is a result of using the prescribed buffer size as the atomic unit upon which to divide iterations. As the buffer size increases, the SPE with the most work can have



(a) Overall profile of hand-coded version

(b) SPE overheads of hand-coded version

(c) Overall profile of Cellgen version

(d) SPE overheads of Cellgen version

(e) Overall profile of Sequoia version

(f) SPE overheads of Sequoia version

Figure 5.6: The impact of DMA buffer size on the performance of each PBPI implementation. *SPE kernel*$_{\max}$ and *SPE kernel*$_{\min}$ show the maximum and the minimum time spent by SPE kernels among 6 SPEs respectively. The total runtime is bounded by the sum of the maximum kernel time and other overheads.

proportionally increasing work. This is the reason that the minimum and maximum SPE kernel time diverge for Cellgen, as shown in Figure 5.6(c).

Sequoia DMA performance benefits from the redundant copy elimination strategy, which avoids unnecessary DMAs when the same data is used in multiple locations. Cellgen obtains a similar benefit for data declared as private; the data is DMAed to each SPE only once.

The multibuffering DMA scheme is used to hide DMA data transfer overhead in all three implementations. However, DMA transfer overhead is exposed when waiting for the completion of the DMA command at the beginning and at the end of the iterations, where there is no computation to overlap with. This overhead becomes more pronounced as the buffer size increases. On the other hand, when computation completely overlaps the data transfer, the major overhead is the cost of checking the completion of a transfer, which decreases as the DMA buffer size increases and the number of DMA calls decreases. When the transfer time of a DMA becomes larger than the computation time for an iteration, it cannot be hidden completely and is exposed as overhead. The DMA wait overhead shown in Figures 5.6(b), 5.6(d), and 5.6(f) includes the cost of checking the completion of DMA transfers and their exposed overhead. The DMA prepare overhead includes the cost of issuing DMA commands and the cost of manipulating buffer addresses and DMA tags.

To optimally run an application like PBPI, it is important to balance the DMA data transfer and computation costs. The cross-over point is reached with different buffer sizes in the three implementations. This difference is due to the variance in the execution time for an iteration in each version. The variance exists because each programming model provides different abstractions which have different associated overheads.

The DMA wait overhead becomes minimal when the buffer size is 2 KB in the hand-coded case, while it becomes so at 4 KB for Cellgen and 8 KB for Sequoia. This discrepancy is due to the difference in the cost of the generated computational kernel and data transfer strategies. Optimal performance is achieved when the sum of the computation costs and all related data transfer overheads is minimal. All data transfer overheads include the exposed wait time for DMAs, time spent to prepare DMAs, and time spent to verify DMA completion. This can be seen in Figure 5.6, as the best performance for each implementation is achieved when the sum of computation costs (*SPE kernel*) and the exposed data transfer overheads (*DMA wait* and *DMA prepare*) are at their minimum. This minimum occurs for the hand-written version at 8 KB, at 2 KB for Cellgen, and at 8 KB for Sequoia.

In the hand-coded case, the epilogue (which includes the computation and communication for the final iterations which are not a multiple of the buffer size) is inefficient: one DMA is issued for each iteration. In Cellgen and Sequoia, one DMA command is issued for the entire remainder of the data. SPE overheads incurred from each DMA transfer decrease as the buffer size increases, and overheads related to an SPE parallel region (such as barriers and signaling) decrease as the total number of parallel regions are called at runtime.

Sequoia has other overheads on the SPE, shown in Figure 5.6(f), including barriers, reduc-

tions, and extra copies of scalar variables which are artifacts of the Sequoia compilation process. Such overheads become noticeable when there is a large number of offloaded function calls. There are 324,071 offloaded function calls in a PBPI run, while there are only 2,592 and 12 offloaded function calls in Fixedgrid and CellStream respectively.

At the end of a leaf task, Sequoia sometimes requires the SPEs to synchronize among themselves for a barrier. In contrast, Cellgen does not require such a barrier among SPEs. Instead, each SPE waits until all outstanding DMAs have completed and then sets a status value in its local storage to indicate completion. The PPE polls these values in each SPE directly, waiting for all SPEs to complete. Cellgen relies on a similar method for collecting the result from SPEs for reduction operations, while Sequoia relies on the DMA and barriers among SPEs. For PBPI, the current Cellgen reduction method is efficient because the reduction data is a single variable. In cases where multiple values are reduced, however, the Sequoia method of using SPEs for reduction operation might be superior. The signaling method used is also different: Sequoia relies on the mailbox communication protocol provided by the Cell SDK, while Cellgen accesses the memory directly. The direct access generally performs better.

5.6 Summary

Shared memory is not the only valid abstraction for a heterogeneous multicore with software-controlled memory. Instead of hiding the fact that the cores are not a part of the normal memory hierarchy, a programming model can represent this separation at the language level. Such a representation can be accomplished by allowing programmers to define tasks that operate only on specified data sets. Such an approach is similar to programming models that abstract distributed programming (such as Charm++ [60], which uses object-oriented constructs to to describe message passing) and to programming models based on task parallelism (such as Cilk [20, 36], which depends on shared memory). Sequoia [34] implements such a programming model. This chapter compared the shared memory approach with the task based, explicit data division approach both in terms of performance and programmability. We found that for data-parallel code, shared memory abstractions with implicit data transfers (such as Cellgen) are more concise and can outperform task based models that require explicit data divisions (such as Sequoia).

This cases study focused on different ways to program a heterogeneous multicore processor with an explicitly managed memory hierarchy. In the next chapter, we look at other kinds of multicore processors.

Chapter 6

Parallel Hardware Architectures: An Evaluation of Shared Memory Abstractions Across the Spectrum

The previous chapter was a programming model comparison with a given parallel hardware architecture, the Cell. We fixed the hardware, and varied the programming models. This chapter is the inverse study: we use programming models designed for data-parallelism, and vary the parallel hardware architectures. For two of the architectures we use a programming model that is data-parallel with shared memory abstractions. For the third architecture we use a programming model that is not strictly a shared memory abstraction, but the parallelization strategy is the same as with the prior models. The hardware we use represents the spectrum of currently available parallel hardware architectures.

There are multiple approaches to designing parallel architectures. Heterogeneous multicores are one such approach, and their potential for high performance is tied to their specialization. Homogeneous multicores have the advantage of relying on existing implementations of shared memory abstractions. The emerging area of general purpose programming on graphics processors (GPUs) presents yet another parallel architecture to explore—its approach is radically parallel compared to all of the previously discussed architectures, yet there are limitations in the kinds of computations that can exploit this parallelism.

We introduce a new context in this study: stream computing. *Streaming* implies the constant arrival of live data which must be processed in real-time. Achieving real-time processing requires both high throughput and low latency. Our work focuses on problems that are relevant to IBM's System S [11, 56, 57, 97, 100] and the Spade (Stream Processing Application Declarative Engine) programming language [38, 46] that runs on top of it. In Spade, *operators* are connected to each other through *streams*. Operators receive, process and send data *tuples* through their streams. Spade is a stream-oriented programming language as streams serve as both the main communication mechanism between operators and determine how an

application is compiled and deployed.

Our goal in this study is to evaluate currently available multicore architectures for accelerating streaming systems. That our study is performed in the context of System S and Spade is important because its streaming nature places constraints on our study. Specifically, we constantly have to move large amounts of data in and out of the accelerator. This requirement will stress a multicore's ability to efficiently transfer data. We choose an operation common in stream computing—aggregation—as a representative task for the requirements of stream computing.

6.1 Motivation

Streaming aggregation is a performance-critical operation in the emerging area of large-scale, distributed stream computing. It is a required operation for any streaming computation that requires data summarization. Further, its salient characteristics—heavy reliance on data transfers, relatively low computation per byte—are similar to other fundamental operations found in stream computing. Hence, accelerating streaming aggregation is an important problem for those that develop and deploy streaming applications and middleware.

In order to attain good performance with an accelerator, developers must first understand how their problem maps to a given architecture. Industry efforts such as OpenCL [61] try to extract a common interface for different multicore architectures. A single interface helps developers because they and their tools can target that interface instead of the disparate architectures available to them. But the abstraction breaks when it comes to performance: different architectures are better at difference tasks and a common interface will not change that.

Our case study investigates how streaming aggregation maps to currently available parallel architectures. We are primarily interested in parallel architectures that are available to developers right now. Multicore architectures are often characterized as "emerging," but that is no longer the case. There are multiple kinds of multicore processors currently on the market. Multicore architectures will certainly continue to change, and perhaps change radically, but developers have to deal with the current multicore reality. The purpose of our case study is to identify which current parallel architectures are acceptable accelerators for streaming aggregation, while at the same time determining which characteristics of our chosen application are applicable to stream computing as a whole.

In our case study, we compare the parallelization of streaming aggregation on three different parallel architectures. We use a sequential version as the baseline. On one end of the multicore spectrum we have an Intel Core 2 Quad system [4], which is a homogeneous multicore similar in principle to an SMP. On the other end we have an Nvidia GeForce GTX 285 GPU [7], which is radically data parallel: thousands of threads performing tiny amounts of work, but with coarse access to main memory. Somewhere in the middle is the

Cell Broadband Engine [23], which is better suited at data parallel computations than a homogeneous multicore [19] but is not as massively data parallel as a GPU. It is better than the GPU at control-intensive code, but not as good as the Intel multicore. Like the GPU, it allows for explicit control of data movement, but like the Intel multicore, it has the same latency and bandwidth connection to main memory.

Streaming aggregation is an obviously data parallel problem that appears often in the domain of high frequency trading [105]. Extracting useful parallelism from the computation is more difficult than it appears due to both its streaming nature and the data characteristics from our domain of high frequency trading. First, its streaming nature means we have only relatively small amounts of data at a time. Second, our data is live stock market trades and quotes [12, 105]. The frequency of trades for a particular symbol roughly follows Zipf's law [22], which causes a severe data distribution imbalance. Using our stock market derived workload, we determine the best configuration for each implementations against each other.

We distinguish our study from prior work in two ways. Two prior studies used code generators specific to their problem domains. The work of Datta et al. [30] used a code generator specific to stencil computations, and the work of Linford et al. [70] used a code generator specific to chemical kinetics. Our study focuses on streaming aggregation, but uses compilers which support a more general class of problems. The second distinguishing characteristic is the class of problems covered by our study. Aggregation performs a single pass over memory, which is in contrast to stencil and chemical kinetics codes which rely on data reuse for high performance. Not being able to benefit from data reuse has a significant impact on an algorithm's suitability to a particular architecture.

The work of van Amesfoort et al. [94] compares the implementation and performance of a data-intensive convolution resampling kernel on platforms similar to our study: a cache-based homogeneous CPU, a GPU and the Cell. Their work looks at a problem that is data-movement bound in a similar way that streaming aggregation is. However, they consider the performance of the kernel in isolation. Because we work in a streaming context, we cannot look exclusively at the performance of our computational kernel. We must also consider the performance of both transferring the data to the kernel, and communicating results back out to the rest of our streaming system. While we are interested in the performance of our computational kernel, we are primarily focused on accelerating streaming aggregation in existing streaming systems. As such, we must consider all data transfer costs associated with real systems.

We set out to answer several questions in our case study. We know that the GPU has enormous computational potential, but do the constraints of streaming aggregation prevent us from being able to exploit it? It is not obvious if the latency and bandwidth between the host and the GPU is sufficient for streaming data. Aggregation is naturally data parallel, and the GPU is computationally better suited for the problem than the Cell architecture, but the Cell has a lower latency access to main memory. Can a lower latency, higher bandwidth connection make up for lack of computational power? Both the Cell and the Intel

Quad communicate with main memory in the same way. However, we can schedule memory transfers on the Cell based on the exact access patterns seen in streaming aggregation. Does such scheduling of memory transfers perform better than the cache-based prefetcher in the Intel Quad? From our experimental results, we answer these questions and derive several lessons:

- GPUs are not well suited to data movement bound algorithms which perform a single pass through memory.
- Fine-grained memory transfers between the host and GPU perform poorly.
- Programmable caches are able to achieve significantly better performance than hardware managed caches with data movement bound problems with regular access patterns.
- In streaming aggregation, data movement efficacy trumps raw computational power.

These lessons provide guidance to those that want to use multicore architectures to accelerate stream computing.

6.2 Background

High frequency trading requires significant computational infrastructure. That infrastructure must be capable of transferring massive amounts of data at high speeds, and simultaneously, perform analysis on that data in real-time. The time requirements are significant, because a late answer is of no use, even if correct.

Large scale, distributed stream computing provides the computational infrastructure and development environment that problems such as high frequency trading require. Questions in high frequency trading that involve multiple trades or quotes of a particular stock are solved with aggregation. The combination of streaming aggregation with stock market data implies two unusual attributes that affect our ability to obtain an efficient parallelization. The first attribute is the inherent streaming nature of dealing with live market data; the second is the frequency distribution of particular stock symbols when dealing with such data. In this section we elaborate on both of these points.

6.2.1 Streaming Aggregation

A streaming aggregation involves at least one stream feeding into an operator where we want aggregate information on some period of history for that stream. For example, a streaming aggregation could be as simple as "for every 5 numbers, emit their average." The *window*

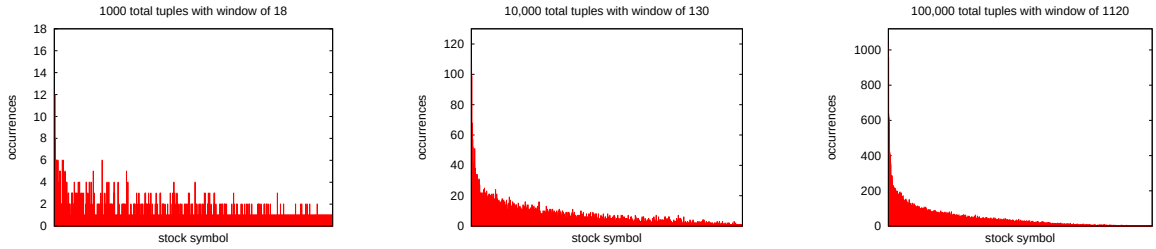


Figure 6.1: Stock symbol frequency distribution histograms.

is the set of tuples involved in each aggregation—in the prior example, the sets are every 5 numbers in the stream. That window is also called a *count-based* window since its size is determined by a count of the number of tuples seen. The alternative is a *time-based* window, where the number of tuples in a window is determined by how much time has passed, which means the number of tuples that will appear in any given window can vary. There are two alternatives for how a window progresses: *tumbling* versus *sliding*. A tumbling window will throw out all of its previous values after each aggregation, whereas a sliding window will slide the window by a predetermined amount. While the computation performed in the example was an average, in principle, the computation can be any that requires a set of values, such as a minimum, maximum or summation.

Aggregations can also be further subdivided into *groups* [12]. Without distinguishing between groups, an operator places all tuples into the same window. When using groups, an operator creates windows for each group class, as specified by the programmer. Our experiments use only count-based, tumbling windows where the groups are stock symbols taken from stock market data. This aggregation is performed when computing the volume-weighted average-price for a particular stock, which investors use to guide their own trades.

Our case study only considers aggregations with count-based, tumbling windows. While the semantics of time-based and sliding windows are different for the consumers of such aggregations, the systems-level implications are similar. Specifically, the memory transfer requirements will remain the same. For this reason, our conclusions should hold for other kinds of aggregations.

6.2.2 Parallel Streaming Aggregation

Inherent in streaming is the concept of tuple ingestion, processing and subsequent generation of the results. This is sequential and predicated on tuple arrival, even though a streaming operator like this can be a part of a larger, distributed parallel application. In order to extract the most parallelism from such a configuration, we must decouple the tuple ingestion from the processing and sending.

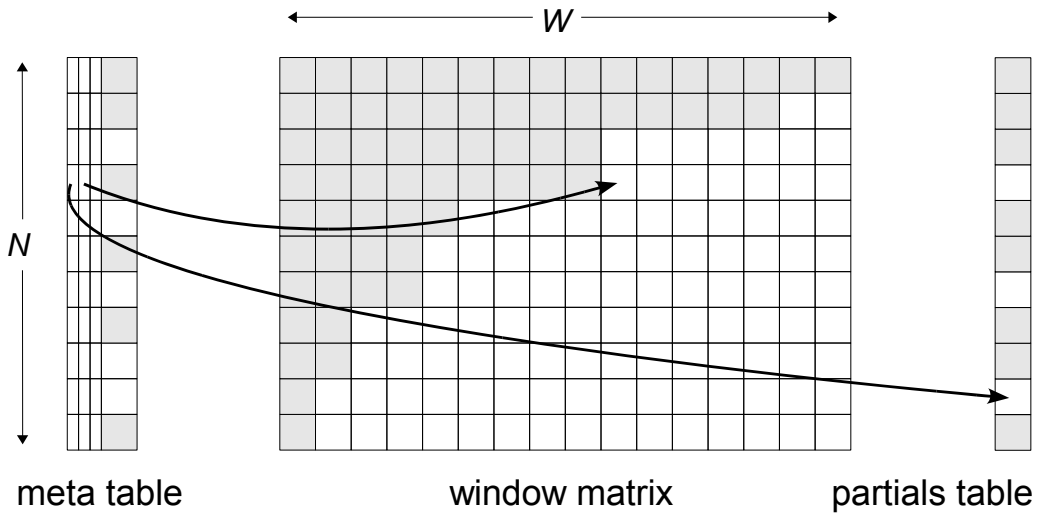


Figure 6.2: Data structures used in partial aggregations.

When the parallel tuple computation has no history—when computing the result for a particular tuple does not depend on any that came before it—the decoupled computation can still occur based on tuple arrivals. But when the parallel computation relies on data history, as it does in the case of streaming aggregations, it can no longer be based solely on tuple arrivals.

We must make sure that each instance of the parallel computation has enough tuples to actually exploit data parallelism. Hence, aggregations become periodic (time-based rather than arrival-based). We have turned a streaming problem into many small batch problems, which introduces a trade-off between low-latency and having enough data to exploit useful parallelism.

Performing aggregations on a periodic basis, instead of when the window is full, requires introducing the concept of *partial aggregations*. Since we will always perform an aggregation at a particular time with whatever data is currently in the window, we need to preserve the partial results so that they can be used during the next time period. The partials table, which contains the partial aggregations, is one of three data structures used in streaming aggregations. Figure 6.2 shows the three data structures and their relationships: the meta table, the partials table and the window matrix. The window matrix is an $N \times W$ matrix where N is the total number of groups in the aggregation (each stock symbol corresponds to a group in our experiments) and W is size of the window. The meta table is of length N, and each entry contains bookkeeping information for a group in the matrix. This bookkeeping information is an index into the partials table, an index for the next available entry in the

window, a flag to indicate if it contains a fully aggregated result, and the result itself (which in our case is the volume-weighted average price).

The meta table must contain an index into the partials table because while the nth entry in the meta table will always map to the nth entry in window matrix, this condition does not hold for the partials table. This discrepancy is due to the fact that the nth entry in the meta table will not always contain the same group. They do not always contain the same group because the meta table and window matrix are populated with tuples from groups in the order they arrive so that the first entries always have windows with at least one tuple. In the partials table, however, the nth entry is always for the same group. This consistency is required because the partials table is used across aggregations, while a single meta table and window matrix are only used for a single aggregation.

In naively constructed data structures, the meta information would be interwoven with the window matrix. However, the memory layout considerations of the parallel hardware we use requires their separation because in certain circumstances we can avoid transferring empty parts of the window matrix.

6.2.3 Stock Market Distribution

Our data is a set of New York Stock Exchange trades and quotes throughout August 8, 2005. This data set contains $N = 2805$ stock symbols and about 12 million trades (18%) and quotes (82%). The frequency of a particular stock symbol being traded in a given day roughly follows Zipf's law, which predicts that frequency of items with rank x appearing is proportional to $1/x^{\alpha}$, where α is close to 1 [22].

In order to reason about our distribution, we appeal to two properties of Zipf's law. First, some symbols will have almost full windows, but most windows will be either empty or have very few tuples in their windows. Second, as this distribution is fractal, the prior point holds no matter what time period we use. We cannot fix the data distribution problem by waiting longer; a longer period will introduce more groups with few tuples. To illustrate this problem, Figure 6.1 shows three different aggregation matrices from three different granularities—waiting for 1,000, 10,000 and 100,000 tuples. Even though each successive matrix contains an order of magnitude more tuples than the next, they all have the general same shape, and with it the same data distribution problems.

6.3 Case Study

Our case study compares the performance of three implementations of streaming aggregation on three different parallel architectures. The parallel architectures represent the current spectrum of multicore processors available to developers. All three implementations

Table 6.1: Hardware used in our case study.

Intel Core 2 Quad	• 4 cores at 2.66 GHz • 8 MB shared L2 • 3.8 GB RAM • 12.8 GB/s max bandwidth from RAM through memory controller • 42.56 GFLOPS
PowerXCell 8i	• 2 Cells at 3.2 GHz • 32 GB RAM • 2 PPEs: 2 SMT threads, 32 KB L1, 512 KB L2 • 16 SPEs: 256 KB local store • 25.6 GB/s max bandwidth from RAM through on-chip memory interface controller • 102.4 GFLOPS
Nvidia GeForce GTX 285	• 240 cores at 648 MHz • 1 GB global memory • PCI Express 2.0 with 16 lanes, 8 GB/s max bandwidth • 1062.72 GFLOPS

require comparable coding effort, leveraging existing compilers and runtimes to produce high-performance code suited to its respective architecture.

The purpose of this case study is to explore the suitability of these parallel architectures for accelerating streaming aggregation. Our end-goal is to allow Spade programmers to mark operators with an *accelerate* keyword. The Spade compiler then generates the correct systems-level code for the desired acceleration hardware. The first step towards this goal is to both identify which parallel architectures can accelerate the computation, and what systems-level code will achieve high performance.

6.3.1 Parallel Hardware

We list the specifications for the hardware used in our study in Table 6.1. The Intel Quad system was also the host for the Nvidia GPU. The physical layout and constraints of our experimental setup are shown in Figure 6.3.

Qualitatively, the Intel Quad is the most general purpose processor, the Nvidia GPU is the most specialized, and the Cell is somewhere in the middle. The Intel Quad is a homogeneous multicore processor with out-of-order execution and a large, hardware controlled cache with hardware prefetching [67]. The Nvidia GPU has 240 cores, where each core has 32 execution pipelines (threads) which have access to 16 KB of shared memory. The execution pipelines inside a core execute in lock-step through the same code. These two architectures represent opposite ends of the spectrum of how to overlap memory latency with computation. The Intel Quad allows a single thread of execution to issue instructions out-of-order. Instructions that cause cache misses do not prevent instructions that do not rely on that data from proceeding. Each instruction pipeline in the Nvidia GPU 285 is in-order, but there are thousands of them, and they can be scheduled in groups of 32 to overcome latency. However, this applies to the Nvidia GPU 285's access to its own global memory. In order to have data, the host must initiate a transfer from main memory, off the motherboard, over a PCI Express bus. Note

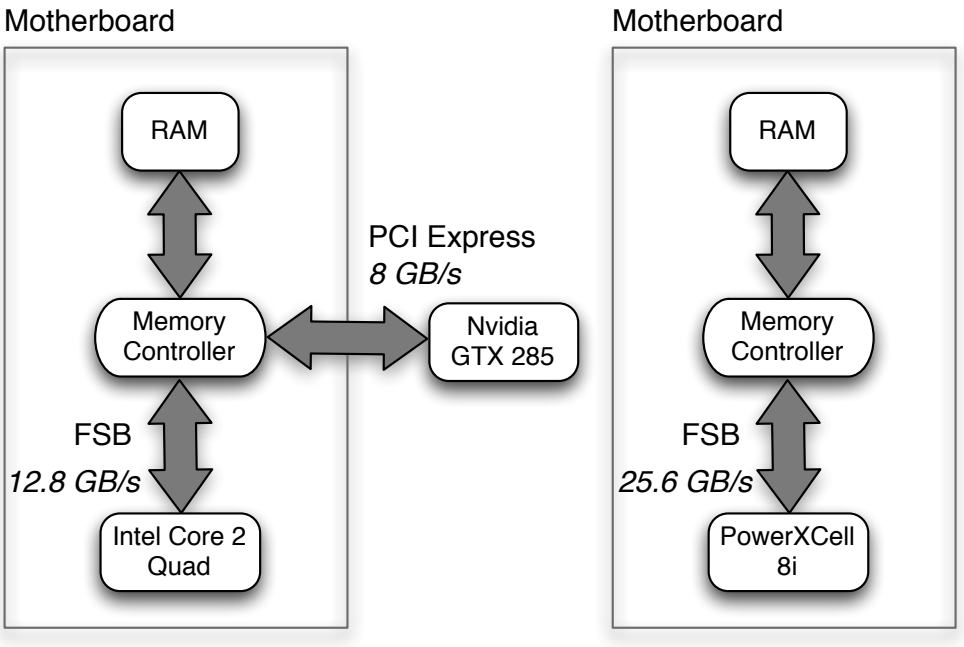


Figure 6.3: Physical layout of our experimental machines. The node with the Intel Quad and Nvidia GPU are on the left, and the Cell node is on the right.

that this means the data must travel through the motherboard's memory controller—just as it must for the Intel Quad—before it travels over the PCI Express bus.

We place the Cell architecture in the middle of these two because it retains direct access to main memory, but it is a heterogeneous architecture suited for data and task parallelism.

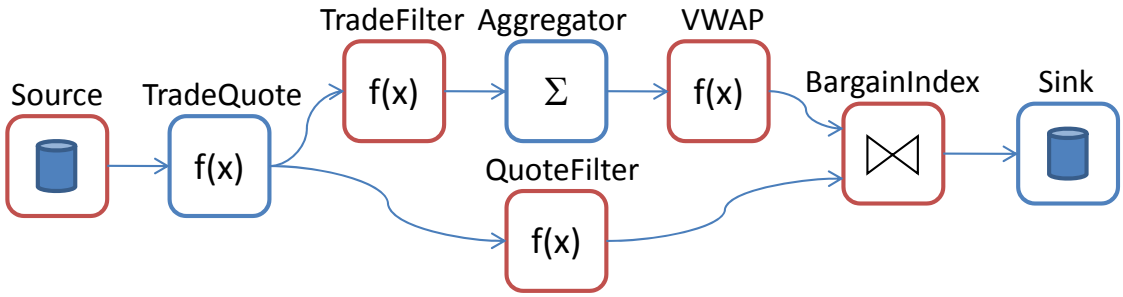


Figure 6.4: Stock market bargain discovery using Spade. Our work occurs in the *Aggregator* operator.

6.3.2 Implementations

We derived our synthetic experiments from the Spade application for stock market bargain discovery depicted in Figure 6.4. The purpose of the application is to analyze live stock market data to discover "bargain" purchases where the current asking price for a stock is less than the volume-weighted average price. The application computes these values by splitting the stock market data into trades (top stream) and quotes (bottom stream). The trades must be aggregated based on their stock symbol over a certain window of time. This operation is a group-based aggregation (see Section 6.2.1). These aggregated values are then used to calculate the volume-weighted average price for each stock. The bargain discovery occurs when the two streams are joined again.

Empirical evaluation has shown that the aggregation operator is the bottleneck in this application. Our benchmark extracts the aggregation and tries to accelerate it by exploiting parallelism. Our experiments use market data from August 8, 2005 to create a statistical model which is used to generate experimental data.

All implementations, whose computational kernels are in Figure 6.5, follow the fork-join model of data parallelism [74]. The program is sequential up until the point of the aggregation. The aggregation is performed in parallel, the details of which depend on the architecture-specific implementation. After the parallel section, the sequential portion of the code has access to the results.

The code we present is systems-level code suitable for the Spade compiler to generate based on high-level Spade programs. Our goal is to protect Spade programmers from having to consider the architectural details of the parallel accelerators available to them.

All implementations are data parallel across groups; aggregations over a particular window happen independently and in parallel. Only groups that have received tuples for a given period will take part in the computation. The data structures involved are the meta table, the partials table and the window matrix, which were described in detail in Section 6.2.1.

```
#pragma omp parallel for schedule(dynamic, 64)
for (int g = 0; g < table->meta->curr; ++g) {
  const int n = table->meta->raw[g].global;

  for (int i = 0; i < table->meta->raw[g].next; ++i) {
    partials[n].vwap += table->matrix[g][i].vwap;
    partials[n].volume += table->matrix[g][i].volume;
    ++partials[n].count;

    if (partials[n].count == W) {
      table->meta->raw[g].rslt.vwap =
        partials[n].vwap;
      table->meta->raw[g].rslt.volume =
        partials[n].volume;

      partials[n].count = 0;
      partials[n].vwap = 0;
      partials[n].volume = 0;

      table->meta->raw[g].send = true;
    }
  }

  table->meta->raw[g].next = 0;
}
```

```
__global__ void aggregatation(AggrMeta* meta,
  AggrPartial (*matrix)[W],
  AggrPartial* partials, const int threads)
{
  const int g = blockIdx.x * THREADS + threadIdx.x;

  if (g >= threads) {
    return;
  }

  const int n = meta[g].global;
  for (int i = 0; i < meta[g].next; ++i) {
    partials[n].vwap += matrix[g][i].vwap;
    partials[n].volume += matrix[g][i].volume;
    ++partials[n].count;

    if (partials[n].count == W) {
      meta[g].rslt.vwap = partials[n].vwap;
      meta[g].rslt.volume = partials[n].volume;

      partials[n].count = 0;
      partials[n].vwap = 0;
      partials[n].volume = 0.0;

      meta[g].send = true;
    }
  }

  meta[g].next = 0;
}
```

```
#pragma cell shared(
  AggrMeta* meta = meta->raw,
  AggrPartial* matrix = matrix[N][W],
  AggrPartial* partials = partials)
{
  int g, i;

  for (g = 0; g < N; ++g) {
    int next = meta[g].next;

    float vwap = partials[g].vwap;
    float volume = partials[g].volume;
    int count = partials[g].count;

    float res_vwap;
    float res_volume;
    char send = 0;

    if (next == 0) {
      continue;
    }

    for (i = 0; i < next; ++i) {
      vwap += matrix[g][i].vwap;
      volume += matrix[g][i].volume;
      ++count;

      if (count == W) {
        res_vwap = vwap;
        res_volume = volume;

        send = 1;
        vwap = 0;
        volume = 0;
        count = 0;
      }
    }

    meta[g].rslt.vwap = res_vwap;
    meta[g].rslt.volume = res_volume;
    meta[g].send = send;
    meta[g].next = 0;

    partials[g].vwap = vwap;
    partials[g].volume = volume;
    partials[g].count = count;
  }
}
```

Figure 6.5: Parallel aggregation kernels. Top left is OpenMP for a homogeneous multicore, bottom left is CUDA for GPUs, and right is Cellgen for Cell. Note that the parallelization effort is similar for all three architectures.

OpenMP

OpenMP [87], a directive-based parallel programming model for C, C++ and Fortran, is well suited for exploiting data parallelism on a homogeneous multicore processor. Each thread in our OpenMP implementation of the data parallel aggregation has the same characteristics of the sequential version. It relies on cache misses and hardware prefetching to move data around the memory hierarchy. Also, it avoids accessing empty values by maintaining groups with non-empty windows in a contiguous portion of the window matrix, as described in Section 6.2.1.

The workload causes a data distribution problem: the stock groups with the most trade transactions in their window are likely to be clustered together. If the window matrix is naively partitioned in contiguous ranges, the thread which gets the beginning of the matrix will have more work than the other threads. To avoid this imbalance, we use OpenMP's dynamic scheduler, which distributes work to threads on-demand as the threads finish their prior assignments. We compiled both the OpenMP and sequential implementations with Intel's C Compiler, version 11 [55].

CUDA

CUDA [6] is an architecture for general purpose programming on GPUs which provides language extensions for C. Note that the code in Figure 6.5 does not include the data transfers from host memory to GPU memory. Before starting GPU computations, we must send both the meta table and the window matrix (Figure 6.2) to the GPU. A separate GPU thread is assigned to each group, and only groups with at least one tuple are sent to the GPU. The algorithm is linear in the number of groups and the threads share no data, which obviates the need to tile global memory access.

After the computation, only the meta table (containing the results) is sent back to the host. The partials table remains resident on the GPU, and the window matrix is no longer needed by the host.

We implemented three versions for the GPU: one which uses synchronous bulk communication between the host and the GPU, one which uses asynchronous bulk communication, and one which does many small transfers. The synchronous implementation sends the meta table and window matrix to the GPU with one memory copy, then waits for the GPU to compute and send the result back to host main memory. However, Nvidia's GTX 285 has a memory controller which can operate independently of the computational hardware. This independence allows our asynchronous implementation to, in principle, overlap GPU computation time with communication time by sending the data for a future aggregation during the computation for the current aggregation.

Both the synchronous and asynchronous implementation send the entire window matrix to

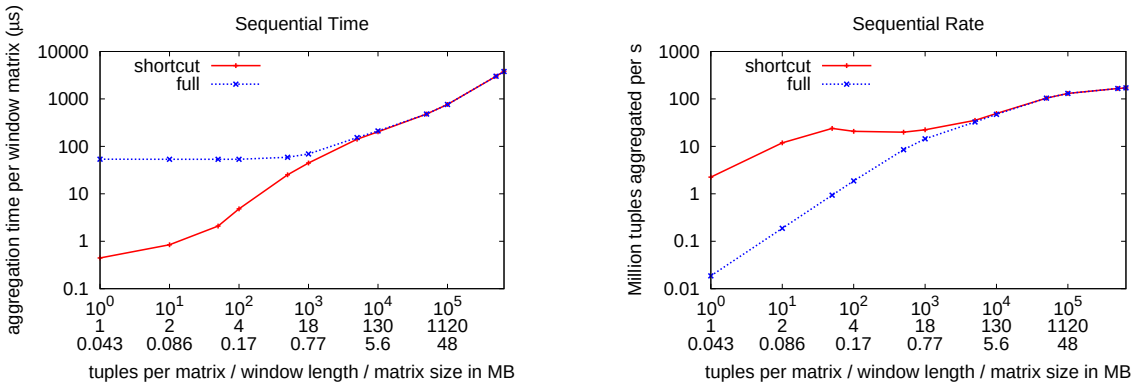


Figure 6.6: Performance of two sequential implementations.

Table 6.2: Data involved in each aggregation.

number of tuples	**1**	**10**	**50**	**100**	**500**	**1k**	**5k**	**10k**	**50k**	**100k**	**500k**	**650k**
window size	1	2	2	4	11	18	49	130	485	1,120	5,000	7,000
meta data (MB)	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11	0.11
window matrix (MB)	0.043	0.086	0.086	0.17	0.47	0.77	2.1	5.6	21	48	214	300

the GPU, even though most of the windows are only partially filled. We allocate the meta table and the window matrix so that they are contiguous in memory, which allows us to issue a single memory copy to transfer all of the data. However, we are still sending data that is not actually used in the computation. Our third implementation does at most $N+1$ memory transfers instead of one memory transfer. The first memory copy is for the meta table, and the remaining transfers send only the actual tuples contained in each window.

Cellgen

Cellgen's programming model and capabilities were discussed extensively in Chapters 2, 3, 4 and 5. Because it provides the means to program a heterogeneous multicore in a similar way that OpenMP can support a homogeneous multicore and CUDA supports a GPU, we use it to implement streaming aggregation on the Cell. We handle the data imbalance by populating the meta table and aggregation matrix in a round-robin manner so that the fullest windows are spread out over the SPEs. Each SPE handles a number of groups determined at runtime.

Over the course of a computation, all of the meta data is transferred to an SPE's local storage. However, if the meta data for a group indicates that the window for that group is empty, that SPE immediately moves on to the next group. Because the meta table and window matrix are accessed in different ways, their transfers are scheduled differently.

Table 6.3: Ratio of data transfer time to total aggregation time for GPU and Cell. For the GPU, timers on the host were used to track the time needed to transfer the data, and then the time needed for the computation itself. For the Cell, timers on each SPE kept track of how much time was spent waiting for a DMA to complete, and separate timers for how long each kernel took to complete.

tuples	1	10	50	100	500	1k	5k	10k	50k	100k	500k	650k
GPU ratio	0.959	0.959	0.961	0.962	0.973	0.982	0.994	0.998	0.999	0.9998	0.99995	0.99997
Cell ratio	0.464	0.467	0.463	0.443	0.372	0.347	0.316	0.317	0.317	0.237	0.172	0.155

6.4 Results

In our experiments, we explored the entire range of number of tuples to aggregate at one time. For example, when we say that a window matrix has 1,000 tuples, that simulates a rate of 1,000 trades a second. The window matrix (of size $N \times W$) is necessarily larger than 1,000 tuples; in this case, $N = 2805$ and $W = 130$. We scaled the window size to match the number of tuples the most populous group contains out of the total 1,000 tuples.

The highest transaction rate seen in our dataset is about 3,000 trades per second. Thus, in our experiments, the range that applies to current market rates is 1,000–10,000 tuples (assuming a one second aggregation frequency). We explored higher rates in anticipation of increased market activity [26], and lower rates to understand at what point parallelization is beneficial. The lower rates are particularly important because they indicate the minimum problem size that can benefit from parallel hardware acceleration. Offloading execution to accelerators and managing parallelism both have an associated overhead. If the time it takes to perform an aggregation sequentially is less than the associated overhead, then that problem size is too small to benefit from parallelism.

An aggregation matrix has dimensions $N \times W$, where N is the number of stock symbols (groups) and W is the size of the window. In our experiments, N is fixed because the number of stock symbols does not change. For each of these experiments we scale the window size, W, so it matches the number of tuples in the most populous group. Due to the nature of our distribution, the most populous group is roughly 1% of the total number of market transactions.

Since W must increase with the total number of tuples in a matrix, the size of the window matrix also grows. Table 6.2 shows the total amount of data involved in each aggregation. All results in this section look at both the execution time for a single aggregation, and the tuples aggregated per second. All scales in our graphs are logarithmic. On the x-axis, in addition to the number of tuples in the window matrix (top), we also label the axis with the window size (middle) and the window matrix size in MB (bottom). This data is in Table 6.2, but we use all three labels to clarify the relationships between performance and the corresponding number of tuples processed, window size and window matrix size.

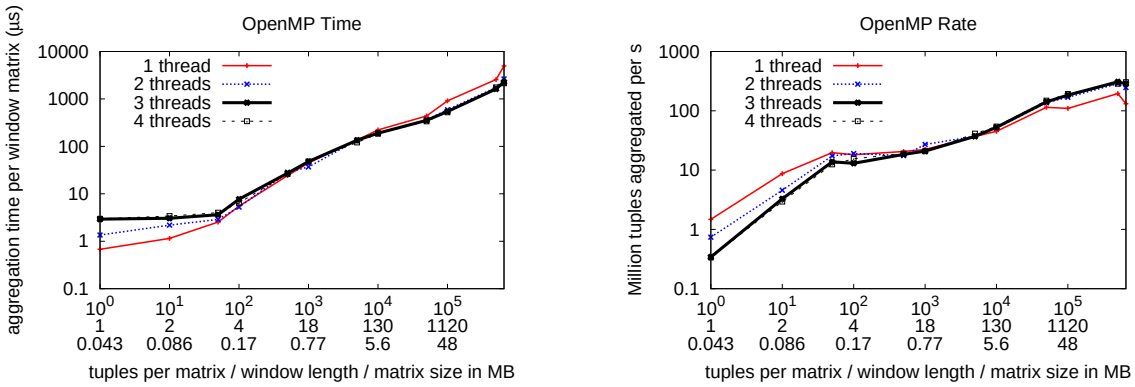


Figure 6.7: Performance of different number of threads used in the Intel Quad implementation.

6.4.1 Intra-implementation Comparisons

Before we can compare the different implementations to each other, we must first establish which parameters are best for each parallel architecture.

Sequential

Our sequential implementation runs on a single core on our Intel Quad node (Figure 6.6). This implementation serves as the baseline. We experimented with two different sequential versions: a *full* method which naively iterates over all N entries of the window matrix, and a *shortcut* method which takes advantage of always packing groups with non-empty windows in the beginning of the window matrix. The *shortcut* optimization is simple, but important: by not iterating over windows that we know are empty, we avoid touching that memory. If we do not access that memory, then we also do not pay the cost of those cache misses. This is an obvious sequential optimization, but the notion of "do not transfer data you do not need" becomes even more important on the parallel architectures.

The difference between *shortcut* and *full* in Figure 6.6 becomes negligible at matrices with 5,000 tuples, which is with a window of $W = 49$. While N is fixed at 2,805 stock symbols for all of our experiments, as the total number of tuples in the window matrix increases, more stocks will take place in the aggregation. Hence, avoiding a linear traversal of N becomes less important.

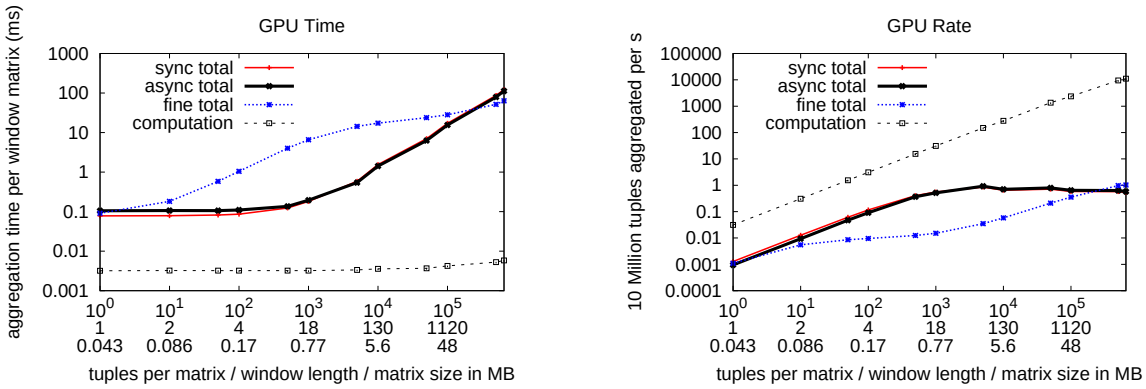


Figure 6.8: Performance of different data transfer strategies for the GPU implementations.

OpenMP on Intel Quad

For the OpenMP implementation on the Intel Quad (Figure 6.7), we varied the number of threads. While the processor has four cores, the shared L2 cache and shared access to main memory inhibit improvement when scaling from three to four threads. As expected, OpenMP with a single thread performs within a close margin, never more than 34%, of the sequential implementation. The difference between the execution of the sequential implementation and OpenMP with a single thread is a measure of the parallelization overhead. The runtime system and synchronization required to parallelize a computation has a cost; there is always a point at which the cost to parallelize a computation is higher than the speedup from working in parallel.

As seen in Figure 6.7, there is no benefit from multithreading until the number of tuples reaches 1,000, where two threads outperform a single thread by 7%. Before this point, the matrices are too small for the work done by each thread to overcome the synchronization costs. After 10,000 tuples, three threads outperform a single thread by a factor of 1.2 to 2.2. The performance of the OpenMP implementation is limited by the fact that aggregations are data movement bound. The hardware limits performance through uncoordinated memory accesses, and a single point of memory access for all threads. The processor requests data from main memory based on cache misses. Since the amount of computation is small compared to data movement time, these cache misses turn into stall cycles.

CUDA on GPU

In Figure 6.8, we compare the synchronous, asynchronous and fine-grained transfer implementations on the GPU. Because most of the cost is in the data transfer from main memory

to the GPU, we also show the time spent only on the computation. All three implementations have the same computation; they only differ in how memory is transferred from the host to the GPU.

The asynchronous implementation makes no more than a 26% difference up until 1,000 tuples, and no more than a 12% difference above 1,000 tuples in total execution time compared to the synchronous implementation. Because the communication is about 100 times more expensive than the actual computation, there is little room for communication and computation overlap. Once the data arrives at the GPU, it is extremely efficient at the computation, which can be seen in Figure 6.8 both by the flat execution time and by the constantly improving aggregation rate. The aggregation is a data parallel problem, and GPUs are efficient data parallel machines. But in this case, the actual performance is not determined by the computation on the GPU, but by the data transfers from host main memory to the GPU. This point is supported by Table 6.3, which shows the ratio of data transfer time to total aggregation time for the asynchronous implementation.

The GPU is also not well suited to many, small memory copies. The implementation with fine-grained memory transfers performs, at worst, 100 times slower than the implementations that do one memory transfer. However, as we increase the number of tuples in each window matrix, which also increases W (the window size), the difference starts to decrease up until 500,000 tuples ($W = 5000$) where the fine-grained implementation outperforms the bulk transfers by 35%. At this point, the window matrix is 300 MB, which is roughly a third the size of the global memory on the GPU. We cannot increase the window matrix significantly and still have enough space for the two window matrices required by the asynchronous implementation.

Cellgen on Cell

We varied the number of SPEs used in our Cell implementation and compared that to the performance of the PPE, as seen in Figure 6.9. We expected the performance of the PPE aggregation to not scale as we increased the total number of tuples. In line with this expectation, even the single SPE aggregation eventually outperforms the PPE aggregation. However, there are startup costs associated with executing a computational kernel on the SPE, and we wanted to identify the cross-over point where the SPE implementations finally outperform the PPE. For all but the single-SPE case, this cross-over is at 100 tuples, which is a matrix of 2,805 stock symbols with 4 trades per window.

That the single SPE case outperforms the PPE at all, which first occurs at 10,000 tuples, is instructive. The single SPE case is not parallel, which eliminates any appeal to simultaneous execution. The aggregation is not computationally bound, so the increased computational power of the SPE does not help. Rather, the single SPE case is able to outperform the PPE because Cellgen generates data transfers based on the access patterns in the code. The PPE and the SPEs use the same memory interface controller to communicate with main

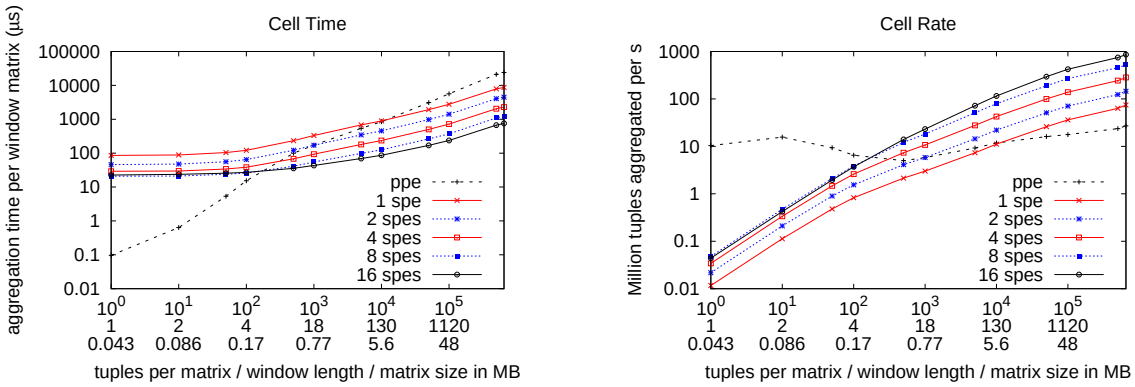


Figure 6.9: Performance of different number of SPEs used for the Cell implementation.

memory. Yet, the PPE relies on cache misses to initiate transfers, while SPEs prefetch data based on memory access patterns recognized by Cellgen. Data prefetching allows for fine-grained overlap of data transfers and computation. Using multiple SPEs introduces parallelism. Hence, the 2, 4, 8 and 16 SPE cases have intelligent, parallel data transfers and scale appropriately. The importance of overlapping data transfers with computation is evident in Table 6.3, which shows the ratio of exposed data transfer times to total time for an aggregation using all 16 SPEs. Comparing the GPU and Cell ratios, unoverlapped data transfer costs account for a significantly smaller fraction of the total aggregation time.

6.4.2 Inter-implementation Comparison

We compare all of the implementations in Figure 6.10, using the best configuration for that hardware as shown by the results in the previous section. For the sequential version, this is the *shortcut* method; for OpenMP, it is with 3 threads; for the GPU we show both the asynchronous implementation with bulk transfers and the fine-grained transfers because there is performance cross-over with large numbers of tuples; and for the Cell it is with all 16 SPEs. The sequential version is our baseline. For small problem sizes, we expect the sequential version to perform the best. The point at which the parallel implementations outperform the sequential version is the minimum problem size needed to exploit parallelism.

In comparison to the sequential version, the OpenMP implementation pays synchronization costs for work distribution and thread coordination. Consequently, parallelism does not help until there is enough work to effectively distribute. This cross-over point occurs at 1,000 tuples in an aggregation. After that point, the benefit of using three cores ranges from 4% to 46%.

The asynchronous GPU implementation never outperforms the sequential version. It suffers

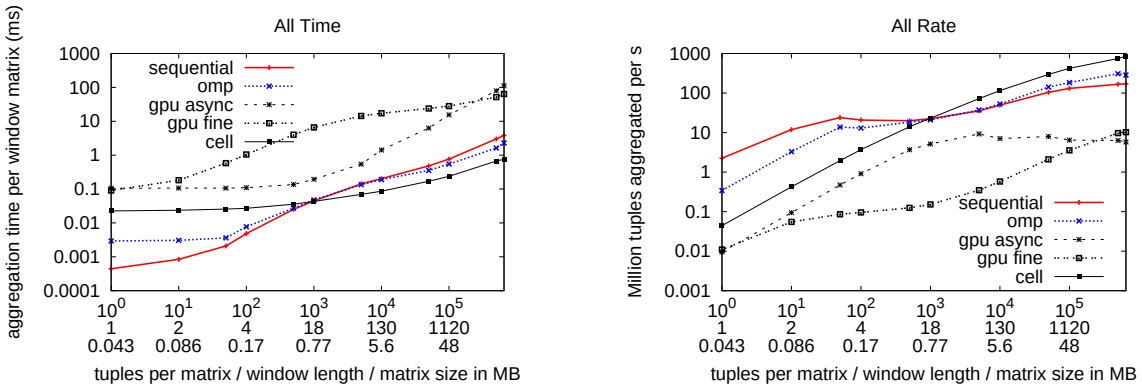


Figure 6.10: Performance comparison of all implementations.

from the fact that it must transfer the entire window matrix to the GPU. In contrast, the sequential version avoids accessing empty parts of the window matrix. Since they are never accessed, the sequential code never pays the cost of transferring data from main memory into the processor cache. The computation on the GPU itself is up to 650 times faster than on a single core of the host CPU, but that is dwarfed by the data transfer cost. Streaming aggregation is fundamentally a data-movement problem, not one of computational power.

While the GPU implementation with fine-grained data transfers eventually outperforms the bulk synchronous implementation, at its best, the fine-grain transfers are still over 10 times slower than the sequential version. The asynchronous bulk implementation tells us that the bandwidth between the GPU and host main memory is too low to overcome the cost of sending the entire window matrix; the fine-grained implementation tells us that the latency is too high to do many, small transfers to avoid sending unneeded data. Future heterogeneous multicore architectures can solve this problem with tight coupling between the main processor and the accelerating co-processors.

The Cell implementation also has startup costs associated with distributing work to the SPEs. The first point at which using the SPEs is beneficial compared to the sequential version is at 1,000 tuples, where the Cell implementation is 3.9 times faster. As the number of tuples increases and the window matrix increases in size, this performance improvement grows to as large as 5 times faster than the sequential implementation.

6.5 Conclusions

Our results show that the Cell architecture is the best fit for streaming aggregation. Further, this result should hold for other streaming operations that perform a single pass through

memory, and have a low computation-per-byte ratio. The Cell architecture fits these class of problems not because of computational power, but data movement efficacy. The GPU is capable of massive data parallelism, but it is not well suited to the many, periodic, small data transfers that are typical in streaming aggregation. Multiple cores of the Intel Quad eventually outperform a single core, but it relies on cache misses to fetch data. The Cell's SPEs have the same low latency, high bandwidth connection to main memory that the Intel Quad has, but the data transfers are based on the access patterns seen in the code, not cache misses. The GPU has more raw computational power than Cell, but it is limited by its connection to main memory. The SPEs can initiate small transfers based on data seen in a computation. In contrast, the GPU cannot dynamically transfer data based on its needs while inside of a computational kernel. Because of this difference, the Cell is able to avoid transferring unneeded data, while the GPU requires it.

Based on these results and experiences accelerating streaming aggregation on three parallel architectures, we draw conclusions for both software developers and hardware architects.

Developers must first understand the memory access patterns in their algorithms in relation to the computation. Our problem, streaming aggregation, is obviously data parallel, but it is not well suited to GPUs, the hardware that is best suited for exploiting data parallelism. Our algorithm performs a single pass of all the memory transferred to the GPU, and only one floating point operation is performed for each discrete value transferred.

Developers must also have an understanding of data movement on the architectures. Homogeneous multicore CPUs transfer data from main memory based on cache misses. GPUs have their own internal memory hierarchy which was not an issue in any of our experiments, but must be well understood to take full advantage of their computational power. On top of that, GPUs have the requirement that all data must first be transferred from host main memory to GPU global memory. The Cell architecture has the same kind of access to main memory as a homogeneous multicore, and with the aid of access-pattern aware compilers such as Cellgen, it can prefetch data.

Developers must finally be able to map their understanding of their algorithms to what will happen on the hardware. Streaming aggregation is not well suited to GPUs because algorithms which perform only a single pass of the transferred data and have little computation per element will not be able to overcome the need to fully transfer all data out of host main memory before the computation starts. Streaming aggregation is well suited to the Cell because its fine-grain data transfers and programmable local store allows prefetching. In contrast, a single-pass algorithm with unstructured accesses to memory would probably perform best on the hardware cache based CPU, and algorithms with quadratic (or worse) memory use would be able to overcome the cost of transferring data to the GPU.

For hardware architects, we appeal to the need for accelerators to be on the motherboard. Our experiments would be different if we had an architecture that was radically data parallel like a GPU, but also enjoyed direct access to main memory. The computational potential for GPUs is extraordinary, but we are limited by the granularity of its memory transfers.

Chapter 7

Related Work

We consider four sets of related work: multicore programming models that implement OpenMP [75, 87] semantics, programming models for multicore processors in general, locality optimizations and transformations and compilation techniques for Cell specifically, and modeling.

Cellgen did not start as an explicit attempt to implement OpenMP on Cell. Rather, Cellgen started as an attempt to codify and abstract existing development practices for Cell. The abstraction of these practices was data parallel code expressed in a shared address space. Once we identified that the existing programming practices for Cell were similar to the programming model behind OpenMP, we then started looking towards OpenMP itself for insight on future directions.

7.1 Programming Models for Multicores

There are many programming models for multicore processors. We start our discussion with those that are most closely related to Cellgen: OpenMP inspired models for the Cell processor. We then extend the discussion to programming models that were not explicitly influenced by OpenMP, and programming models whose implementation was not focused on Cell.

7.1.1 OpenMP and Related Models for Cell

The projects most directly related to Cellgen are the efforts to support OpenMP on Cell coming from IBM [24, 25, 32, 33, 86] and joint efforts between IBM and the Barcelona Supercomputing Center (BSC) [15, 17, 95]. The most recent effort from IBM is the Direct Block Data Buffer (DBDB) [72] project. Its approach to supporting a high performance,

shared memory abstraction on the Cell is similar to Cellgen: use multi-level buffering for predictable memory access patterns. It is in the context of a full OpenMP implementation, so it defaults to a software-cache for irregular accesses. However, DBDB has no support of the kind of stencil support that is allowed in Cellgen; for off-induction accesses, it does not use a rotating system of buffers to re-use the buffers in stencil accesses. DBDB also takes a different approach for how to handle accessing strided data in memory. At runtime, DBDB figures out the smallest amount of data it can transfer for each DMA list element that contains each strided element in memory—this is the *transfer layout*. Once the data is transferred to the SPE, it is copied to an *access layout*. The access layout stores the data elements contiguously, and the data that was picked up as padding is stored at the end of the buffer. If the elements are written to, then they are copied back to the transfer layout before the DMA to main memory. In contrast, Cellgen assumes that the data is padded in main memory—a transformation that it can perform automatically in the future. This difference means that when accessing strided data, Cellgen can avoid extraneous copies at runtime. Their paper also has a major methodology difference in that they only compare to software-cache implementations. In contrast, Cellgen's performance evaluation has focused on comparing against expertly tuned applications. Cellgen was originally published in February, 2009 [91], and DBDB was originally published in June, 2009 [72]. Finally, the implementation of DBDB is unavailable for a direct comparison with Cellgen.

The other project from IBM to support OpenMP on the Cell is described in its first version in Eichenberger et al. [32, 33]. The earlier publications by Eichenberger et al. discuss the entire Cell compiler framework as shipped with IBM's SDK. This discussion includes code generation optimized for the SPE architecture, paying particular attention to handling branches and SIMD code. However, they do also discuss an implementation of OpenMP for Cell that relies on a software cache. The work presented by Chen et al. [24] and O'Brien et al. [86] are an extension of the software cache version of OpenMP to allow for the buffering schemes similar to Cellgen and DBDB. However, this work does not support the kinds of stencil accesses presented in Section 3.7.

Much of the work by researchers at IBM and BSC has focused on supporting OpenMP's shared memory abstractions through optimizing software caches. As discussed in Chapter 4, the work of Chen et al. [25] attempts to improve software caches by prefetching irregular references. This improvement is accomplished by executing a provisional version of each loop that collects and subsequently prefetches the addresses that irregular accesses require. When the computation version of the loop is executed, those accesses are much more likely to result in a hit. The work of Vujic et al. [95] presents a software cache that differentiates between high and low locality accesses—something that can be recognized at compile-time through access analysis. High locality accesses can benefit from prefetching, greatly increasing the likelihood of cache hits. Low locality accesses are unlikely to result in cache hits, and their irregularity means that prefetching is difficult. Such accesses are directed to a cache which has a lower miss penalty at the expense of a lower hit rate.

There is a spectrum of design approaches to supporting shared memory abstraction on the

Cell. On one end of the spectrum lie software caches with no optimization. This approach supports everything, but at the expense of performance. The software cache described in Eichenberger et al. [32] represents this approach, and the results in Section 3.7 demonstrate its performance limitations. For the microbenchmarks in Section 3.7, the software cache performance is at best 2 times worse than Cellgen, and at worst, 31 times worse than Cellgen.

Cellgen as it was presented in Schneider et al. [91]—no column accesses, and no stencil support—represents the opposite end of the spectrum. This approach was to only support the kinds of accesses that, at the time, we knew how to generate high performance code for. The disadvantage of this approach is that we were limited in the kinds of codes we could support. Iterations on the designs at both ends of the spectrum have shortened the gap. The previously mentioned software cache optimizations take advantage of the unique situation presented by a software cache: the compiler can not only influence cache behavior with respect to particular accesses, it can outright control it. Cellgen's support for stencil accesses relaxes the prior requirement that all loop iterations must be independent, and it has none of the overheads associated with a software cache. DBDB is the project that is the closest to being in the middle of the spectrum, with direct use of multibuffering and software cache for irregular accesses. But, no single project incorporates all of the optimizations. Such a design would have:

- Direct multibuffering for predictable memory accesses, similar to Cellgen and DBDB.
- Direct multibuffering for stencil accesses, similar to Cellgen.
- A software cache for all irregular accesses—but the software cache must not interfere with multibuffering.
- Prefetching irregular accesses, as described by Chen et al.
- All of the above may preclude the need for distinguishing between high and low locality accesses to the software cache—most, if not all, high locality accesses should be managed with direct multibuffering. But if the distinction still exists, the best solution should distinguish between high and low locality accesses to the software cache, as described by Vujic et al.

Another project from BSC influenced by OpenMP is Cell Superscalar (CellSc) [17]. Like OpenMP, CellSs augments sequential code with pragmas that instruct a compiler what to parallelize. The pragmas mark a function that should be parallelized across the SPEs, along with annotations for whether variables are used in an input or output context. CellSs is implemented in a compiler framework that was originally targeted towards superscalar processors. The authors use this prior work to maintain a directed dependency graph at runtime. The nodes in the graph are tasks to be executed on the SPEs, and the edges

represent data dependencies. The CellSs runtime will schedule independent tasks on separate SPEs and subgraphs of dependent tasks on the same SPE to take advantage of data reuse. The approach taken by Cellgen differs in two ways. First, Cellgen operates at the level of arbitrary code blocks, not functions. Transformations such as those required by handling stencil accesses would have to be performed by hand by the programmer; only the innermost computation is marked for offloading to the SPEs. Consequently, transformations that need to apply to any code outside of an inner loop are not possible. This difference, however, is related to the second major difference: Cellgen is primarily concerned with data parallelism, and CellSs is primarily concerned with task parallelism. (However, their model can be readably adapted to data parallelism, just with the prior limitations.) As a consequence of this focus, Cellgen uses all SPEs for each offloaded region of code. CellSs allows multiple inflight tasks and does not use all SPEs for each.

Recent efforts for extending OpenMP with directives that manage dependent tasks [31] are directed towards improving locality by automatically managing dependencies, hence data transfers, between tasks executing on different processors or accelerators. Cellgen takes a different path, by managing locality and communication overlap through implicit task and data decomposition.

7.1.2 General Multicore Programming Models

The Accelerated Library Framework (ALF) [27, 51] is a library designed for developing applications on accelerator based architectures. The API is designed so that it can be applied to general accelerator-based multicore architectures, but the current implementation is only for Cell. Unlike high-level programming models like OpenMP, Cellgen or CellSs, ALF is a library only. It is closer to the runtime systems that such programming models depend on. ALF abstracts various tasks—such as specifying data working sets, determining what level of multibuffering to use and starting accelerated functions—from the underlying hardware. But the programmer must still explicitly invoke these actions at the appropriate time.

Sequoia [34] was covered extensively in Chapter 5. The main concept behind Sequoia is to develop applications with a multicore processor's memory hierarchy in mind. Programmers develop algorithms with abstractions that enable them to define how data should be decomposed, and what computations to perform on that data at each level of decomposition. The Sequoia runtime is then responsible for determining how to map these data decompositions to the target architecture's memory hierarchy. Alternatively, Cellgen provides a shared memory abstraction on the Cell architecture. The performance benefits of Cellgen over Sequoia were presented in Section 5.5.

RapidMind[1] [77, 78] is a language and development platform for multicore processors that is

[1]RapidMind (the company) was purchased by Intel in 2009 [84], and RapidMind (the technology) was

independent of the OpenMP programming model. Similar to OpenMP, RapidMind extends an existing language (C++) with parallel constructs. Different from OpenMP, RapidMind's extensions are richer in both semantics and scope. OpenMP extends C, C++ and Fortran through pragma directives. Two styles of parallel programming are possible with OpenMP: data parallel and task parallel. For both styles, programmers write sequential code, then augment their code with OpenMP directives to parallelize it. If programmers remove all of the OpenMP directives from their application, they are left with a valid sequential program. This invariant does not hold for RapidMind—its extensions change the way programmers will develop their applications.

RapidMind's functionality is provided to programmers through its data types and program objects. Programmers can define multidimensional arrays of value types. Value types represent N-tuples of the native numerical types. Of course, C++ programmers already have the ability to define such constructs. The RapidMind data types, however, are more than just abstract data types. Their use enables compile time and runtime optimizations such as generating SIMD instructions and avoiding unnecessary copies. Program objects define computations on the data types. Inside program objects, programmers must use RapidMind specific macros for basic control structures such as if-else constructs and looping. These macros enable compile-time, architecture specific optimizations. Program objects are defined from the perspective of a single element—the inner-most loop in conventional data parallel code. (This perspective is similar to programming for GPUs using CUDA.) In order to initiate accelerated computations, programmers apply program objects to arrays. Because the program objects are defined to operate on a single element of the array, the compiler and runtime system are free to parallelize the operation in a manner that best fits the current architecture.

Program objects are similar to closures in functional programming languages: they can close over local variables; they allow currying (partial application); and they can be composed with other program objects to create new program objects. These semantics are richer than those provided by Cellgen and the OpenMP programming model in general. However, RapidMind's semantics are expressed in a significantly more cumbersome syntax. One advantage of Cellgen's approach is that programmers do not need to adapt their applications to a new style of programming.

EXOCHI [96] enables OpenMP for heterogeneous multicores, but it does so in a radically different way than the previous projects. EXOCHI is an extension of the Multiple Instruction Stream Processor (MISP) [43] project, which supports shared memory programming on heterogeneous architectures by extending the instruction set and exposing architectural resources directly to application level, bypassing the operating system. EXOCHI builds on this infrastructure by allowing "fat" binaries, where instructions for non-IA32 accelerators can live alongside IA32 instructions. Non-IA32 cores can share an address space with IA32 cores through address translation support at the architectural level only, without requiring

renamed Intel Array Building Blocks [3].

modifications to the operating system.

Because EXOCHI has architectural support for shared memory, it is able to support Pthread-style multithreading across heterogeneous cores. Implementing OpenMP on top of such architectural support has similar complexity to implementing OpenMP on top of a typical SMP. With compiler support for such an implementation of OpenMP, legacy applications can execute on heterogeneous multicores. Cellgen's approach is to support an OpenMP style of programming by managing data transfers itself. The EXOCHI approach is to try to provide a shared memory abstraction even at the instruction level.

Streaming languages [29, 39, 42] also expose data locality to the programmer via the stream abstraction. Decomposing data streams into input/output blocks and buffering these blocks in local memories is the equivalent of decomposing loops into tasks and scheduling the transfers for the input/output sets of each task in Cellgen. Earlier studies on streaming languages for both conventional and streaming processor architectures [29, 42] have demonstrated that the stream abstraction enables locality optimization via compiler/runtime support. We make similar arguments for Cellgen: promoting programmability without performance penalty.

Another attempt to abstract away difficulties of programming for heterogeneous multicore processor is Hera-JVM [79, 80, 81]. Instead of introducing new abstractions, or supporting high performance extensions such as OpenMP, Hera-JVM is a Java virtual machine implemented specifically for the Cell. Programmers can provide hints to guide mappings of Java threads to the SPEs or to the PPE. Data transfers to the SPEs' local stores are accomplished through an object cache, but it has no coherency among the SPEs. Local changes in an SPE are committed globally at synchronization points.

Charm++ [60] is a distributed, object-oriented programming model based on C++. Recent efforts have extended Charm++ to also work on the Cell [65]. The result is a distributed programming model where programmers tag certain Charm++ functions with an accel keyword to indicate that the task should run on a Cell processor. Such functions are called accelerated entry methods. Similar to Sequoia, programmers need to specify how the variables are used in the accelerated entry methods. A specialized runtime [64] is responsible for telling the SPEs which accelerated entry method to execute, transferring the data to the SPEs before the execution of an accelerated entry method, and transferring the results back from the SPEs after its completion. This runtime is similar in design to the runtime that Cellgen uses, which is a direct extension of the runtime developed by Blagojevic et al [19]. However, there is no library or compiler support for multilevel buffering on the SPEs. Programmers who are implementing data-parallel code can circumvent this fact by only making the inner-loop of a computation an accelerated entry method. This technique, however, is potentially cumbersome to program, and detrimental to performance since the PPE must be involved after each iteration of the inner loop.

The Liquid Metal project as presented by Huang et al. [49] represents a more radical approach than others. While it does use object-oriented programming as a basis, its goal is to find a high-level programming model that can describe computation on many different ar-

chitectures: a virtual machine on a conventional processor, Cell, GPUs and FPGAs. Huang et al. present Lime, a language that extends Java to allow value types that have precise semantics which can easily map to hardware. They also have a corresponding compiler infrastructure which can produce either Java byte-code for a Java Virtual Machine on a conventional processor, or code in a hardware description language suitable for an FPGA. While the design they present focuses on how to map computations to an FPGA, they are also targeting other multicore architectures such as the Cell and GPUs. Similar to Sequoia or Charm++, Liquid Metal is an attempt to allow programmers to write a computation in a high-level language that can be compiled to multiple architectures. Targeting FPGAs, however, imposes the additional burden of segregating stateful and stateless code. Cellgen and projects like CellSs and DBDB do not attempt to create a new programming model, but to map an existing programming model onto new architectures.

7.2 Low-Level Compiler Optimizations

Cellgen is a source-to-source compiler. Since it generates C code, it depends on an optimizing compiler to generate high performance assembly code. The focus of this work has been on the burden imposed by having heterogeneous cores with separate memory hierarchies, but the SPEs themselves present new challenges for low-level code generation. As such, the work described by Eichenberger et al. [32, 33] that pertains to subword optimization, branch optimization and instruction scheduling could directly benefit applications that use Cellgen. The work of Knight et al. [62] focuses on the intermediate representation in the compiler, sitting somewhere between the level that Cellgen operates on and the work of Eichenberger et al. They design and implement an intermediate representation (IR) for architectures with explicitly managed memory hierarchies, where parallelism and data transfers are explicit in the IR. Their optimizations focus on transformations to this IR code. This work has also considered auto-vectorization outside of its scope, although experiments in previous chapters have demonstrated that vectorized code can significantly improve performance. The work of Nuzman et al. [85] describes techniques to auto-vectorize operations on data that is not already compactly stored in memory. They describe techniques to recognize and automatically vectorize computations that operate on elements that are not contiguous in memory. If such work was coupled with Cellgen, it would enable both high performance memory access patterns and high performance computations.

7.3 Locality Optimizations

There is a significant body of work for improving the locality of loop-based codes through code transformation [66, 68, 69, 98]. These techniques change the iteration space of the loops from accessing complete rows or columns at a time to accessing sub-matrices (also known as

blocks or *tiles*) so that data is reused in the cache.

Cellgen is concurrent to parallelizing and optimizing compiler frameworks that seek to extend such efforts to recent multicore processors by optimizing data movement to and from explicitly managed memories. Recent work which focuses on scratch-pad memories that can sustain cache misses tries to distribute loop iterations to cores based on their memory access patterns in order to ensure a balanced computation [102, 103]. The polyhedral model for loop-nest optimization has been extended for GPUs [16]. Similar to compilers based on the polyhedral model, Cellgen targets locality by blocking data references, however Cellgen focuses further on the scheduling and optimization of data transfers to maximize latency overlap. In addition, Cellgen targets optimization of explicitly annotated, OpenMP-style parallel code rather than full automatic parallelization. It is capable of compiling, with reasonable efficiency, large scientific codes.

7.4 Modeling

Cellgen uses a model to predict the runtime of offloaded computational loops. This model is a descendent of the LogP [28] family of models originally described by Culler et al. The original LogP model recognized the importance of communication overhead and latency in estimating the execution time of a parallel application. Communication time is important because of the trend that in parallel computation that the authors recognized: many computational nodes connected on a network, where each computational node has its own processor, memory and disk. LogP uses the following parameters:

L: The *latency* of communicating a message of size w from one node to another.

o: The *overhead* paid at a node for the computation time needed to send or receive a message of size w. Note that when one message is sent, the sending node must spend o time prepping it, and the receiving node must also spend o time receiving it.

g: The time *gap* a node must wait between sending subsequent messages.

P: The number of *processors* involved in the cluster. Since the model assumed that there would be one processor per node, this is also the number of nodes.

The work of Alexandrov et al. [10] extends the model with one more parameter:

G: The *Gap per byte* of long messages.

The resulting model is called LogGP. The LogP model has an implicit parameter, w, which is the size of small messages. Alexandrov et al. recognized that many nodes had specialized

transfer mechanisms for long messages, and if these were not modeled, then the predictions could be inaccurate by wide margins. The work of Bosque and Pastor [21] further extended the model to HLogGP: LogGP for heterogeneous clusters. Bosque and Pastor recognized that there is a tendency towards heterogeneous clusters where individual nodes may significantly different computational power. Their model uses a vector or matrix as appropriate as a replacement for the scalar values in the prior models. This change is necessary because once the cluster is heterogeneous, all of the parameters may be different depending on which nodes are involved in exchanging messages. In particular, P is no longer just the number of nodes in the cluster. Rather, $\mathbf{P}$ is a vector which represents the relative computational power of each node.

The work of Blagojevic et al. [18] is a model for multiple levels of parallelism on heterogeneous multicore processors. While the model is focused on a single node, the models described above apply since the advent of multicore processors means that a single processor has similar characteristics to a cluster of conventional processors. They evaluate their model using a Cell processor. Clearly, their model has the most direct relationship with the model presented in Section 3.9. However, their model's purpose is to predict the execution time of a whole application. The model presented in Section 3.9 is intended to predict the execution time of a single computational loop. As such, it makes finer considerations, such as which one of the dual-issue pipelines dominates in a computation.

Chapter 8

Conclusions

The Cell is a dying platform [13]. While this work is about heterogeneous multicore processors in general, it is unavoidable that much of the effort has gone into supporting a particular instance of such a processor. The obvious question, with respect to the death of the most visible instance of a heterogeneous multicore processor, is what is the continued relevance of this work?

The first reason for continued relevance is that the Cell may be dying, but heterogeneity is not. The design lessons from the Cell may be integrated into the mainline PowerPC architecture [101]. Further, the Cell is not the only instance of heterogeneity; Larrabee [92], Pangaee [99], AMD Fusion [1] and most recently, ARM's Mali [5] are all heterogeneous processor architectures. The arguments for heterogeneity presented in Chapter 1 still hold, and we will likely see more heterogeneous processors in the future. The techniques and insight provided by this dissertation will help manage the inherent difficulties of programming such heterogeneous processors.

The results in Chapter 6 make a further argument for heterogeneity. As shown through the experimental evaluation comparing a heterogeneous multicore, a homogeneous multicore and a graphics processor, the raw computational power of the graphics processor is enormous. Its limitation was its communication to main memory. The current state of graphics processors is perhaps a stop-gap solution. The obvious solution to alleviating the memory latency problem is to fully integrate a graphics processor on the same chip as the host processor. This design, however, is a heterogeneous multicore processor.

Further, the fundamental approach this dissertation takes to supporting a shared memory abstraction on a heterogeneous multicore can have application to homogeneous multicores. The fundamental approach is to identify memory access patterns at compile time, and to generate data transfers that enable high performance based on those access patterns, tailored to the architecture. Current homogeneous multicores have hardware controlled caches. The first level of these caches, however, is private to each individual core. These hardware con-

trolled caches could expose software-level mechanisms to influence which data is brought into the private caches. If such mechanisms are exposed, then the static analysis and code transformations presented in this dissertation could apply with little change. While such analysis and transformations would not be required for correctness, they can increase performance.

We can take this reasoning higher up the system stack. Consider implementing a shared memory abstraction over a cluster of compute nodes. Each compute node would map to an individual core in a heterogeneous processor, and the network connecting the computer nodes would map to the interconnection network in the processor. The bandwidth capabilities and latency overhead of the network connecting nodes in a cluster will be significantly different than the interconnection network on a chip. Because of the connection differences, the nature of the transformations may be significantly different—the need for a performance model to inform runtime decisions about the efficacy of parallelizing computations would surely increase. But the same fundamental analysis would have to take place: what are the memory access patterns, and how can that data be efficiently transferred to and from the computational elements.

We can even relax the assumption of static code analysis—fundamentally this work is about *recognizing* memory access patterns. Static code analysis is one such means of achieving this. Such recognition could happen in the context of a virtual machine, in which the runtime system could have knowledge of the actual addresses being accessed as they are accessed. Such a capability could find regularity in the kinds of accesses that must be classified as irregular by static analysis. Finally, an optimizing just-in-time compiler could generate the data transfers based on the runtime access analysis.

The other purpose of this work is to inform processor architects of the utility of their designs. One of the reasons that the Cell processor itself will not survive in its current state is the inherent difficulty in programming for it. While heterogeneity will necessarily introduce complexity, some of complexity of the Cell was perhaps unnecessary. One of the purposes of this work was to hide the architectural details of a heterogeneous processor to the programmer. Our goal was to always maintain high performance, but even basic correctness took a significant amount of time to achieve. As explained in in Chapter 7, a complete solution for supporting a shared memory abstraction on the Cell will eventually use a software cache as a fallback. However, as demonstrated in Chapter 4, the software cache itself is at a significant performance disadvantage. These two results present a situation where we know we will eventually need something that does not achieve good performance. However, we can avoid this situation if the architecture itself had a hardware cache.

Arguing for a hardware cache is perhaps antithetical to the design philosophy of the Cell processor. However, it is not antithetical to the design of heterogeneous multicore processors in general. We should use the lessons of past architectures to inform how to design future architectures. One such lesson is that without a hardware cache, we will eventually want to fall back on a software cache for some kinds of irregular accesses. Consequently, integrating a simple hardware cache that still allows for explicit software control could alleviate this

problem. The work in this dissertation would still be applicable to such an architecture, with one major difference: it would only be necessary to achieve high performance. Basic correctness would be supported by the hardware. As a consequence, development for such an architecture would be significantly easier because developers could write code that is correct but not efficient, then incrementally improve performance. With the current Cell architecture, achieving basic correctness is in the same order-of-magnitude of effort as achieving high performance.

The final issue is the most fundamental: the expression of parallelism. How we handle parallelism at the language level may become one of the most important issues in Computer Science in the coming decades. Parallel programming techniques have long been used by the high performance community to achieve fast, scalable applications. Mainstream programmers have used concurrency primarily for software engineering purposes. They may use multiple, communicating threads and processes, but these techniques are used because they are the most natural way to structure their application—not to achieve high performance. The advent of ubiquitous multicore processors is the catalyst for forcing these techniques from the relatively small community of high performance computing to the mainstream use of most programmers.

Programming abstractions become mainstream when two conditions are met: when they are affordable, and when they solve a pressing problem. Historically, this trend has happened several times. The first compiler was completed in 1957 for Fortran [14], but it was many years before high-level, compiled languages replaced assembly as the dominant means to program computers. The sophistication of compiler optimization techniques took time to mature to the levels such that the gains over programming in assembly were marginal in most areas. At the same time, the need for portable programs arose when the complexity of developing exclusively at the assembly level for each architecture became too large.

Garbage collection was first used in Lisp in 1960 [76]. While garbage collection has been used in many programming environments since then, it did not become a part of mainstream programming until after the introduction of Java in 1995 [40, 41]. By late '90s, computer speeds were such that the performance degradation from garbage collection was becoming preferable to manual memory management, which is notoriously error-prone. Since then, improvements in both virtual machines in general and garbage collection specifically have maintained the economics of that trade-off.

The rise of the World-Wide-Web has seen, along with it, the rise of scripting languages. Perl was first born as a replacement for shell scripting and stand-alone Unix utilities in 1987. But it was its proliferation as a web programming language that Perl entered mainstream usage. In the recent decade, other dynamic scripting languages such as Python and Ruby have taken on the same role. These languages use dynamic runtime typing, garbage collection on top of a virtual machine, and generally execute slower than the compiled alternatives. Yet, these languages remain popular for web programming because the alternatives (C, C++, Java, etc.) are considered too cumbersome and verbose for the relatively high level concerns of a

web developer, and the computational power of today's machines can easily handle them.

Parallel programming has primarily been the domain of high performance computing. Mainstream parallel programming has long been possible, but is has never been necessary. The rise of multicore processors is finally making the long-predicted necessity of mainstream parallel programming a reality. However, unlike prior programming abstractions that had a performance *cost*, parallel programming techniques are a performance *enabler*.

It is the job of the high performance computing community to assess the applicability of our current parallel programming techniques to the newly emerging multicore processors. Techniques that abstract the underlying hardware as much as possible are more likely to be adopted by the widest community. Therefore, we have to assess both the programmability of these techniques, and demonstrate their ability to achieve high performance. If a parallel programming abstraction is unable to outperform sequential execution by a significant margin, then it is not worth the programming effort.

This dissertation is part of the process of evaluating the applicability of our existing techniques for parallel programming to new multicore processors. The shared memory abstraction presented in this dissertation may not be widely applicable; mainstream programmers may not have a use for it as data-parallelism is a subset of parallel programming in general. More general models have recently found traction in recent languages, such as Communicating Sequential Processes (CSP) [47] in Go, the Actor model [44] in Erlang, and transactional memory in Clojure. However, evaluating shared memory based programming models is still a necessary step in the overall area of expressing parallelism. This dissertation contributes such an evaluation. Shared memory abstractions are suitable for heterogeneous multicores in terms of both programmability and performance. Future developments and needs will determine if the programming models explored in this dissertation are applicable to a wider audience.

Bibliography

[1] AMD Fusion. http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx.

[2] Boost.Spirit. http://spirit.sourceforge.net.

[3] Intel Array Building Blocks. http://software.intel.com/en-us/articles/intel-array-building-blocks.

[4] Intel Core 2 Quad. http://www.intel.com/products/processor/core2quad.

[5] Mali Graphics Hardware. http://www.arm.com/products/multimedia/mali-graphics-hardware/index.php.

[6] Nvidia CUDA. http://www.nvidia.com/object/cuda_home.html.

[7] Nvidia GeForce GTX 285. http://www.nvidia.com/object/product_geforce_gtx_285_us.html.

[8] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *Commun. ACM*, 53:90–101, August 2010.

[9] T. Ainsworth and T. Pinkston. On characterizing performance of the cell broadband engine element interconnect bus. In *NOCS 2007: First Internationl Symposium on Networks-on-Chip, 2007*, pages 18–29, May 2007.

[10] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long messages Into the LogP Model—One Step Closer Towards a Realistic Model for Parallel Computation. In *SPAA '95: Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, New York, NY, USA, 1995. ACM.

[11] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. In *ICDCS '06: Proceedings of the 26th IEEE International Conference on Distributed Computing Systems*.

[12] H. Andrade, B. Gedik, K.-L. Wu, and P. S. Yu. Scale-up Strategies for Processing High-Rate Data Streams in System S. In *ICDE '09: Proceedings of the IEEE 25th International Conference on Data Engineering*.

[13] Ars Technica. End of the line for IBM's Cell. http://arstechnica.com/hardware/news/2009/11/end-of-the-line-for-ibms-cell.ars, November 23, 2009.

[14] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt. The FORTRAN Automatic Coding System. In *Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability*, IRE-AIEE-ACM '57 (Western), pages 188–198, New York, NY, USA, 1957. ACM.

[15] J. Balart, M. González, X. Martorell, E. Ayguadé, Z. Sura, T. Chen, T. Zhang, K. O'Brien, and K. M. O'Brien. A Novel Asynchronous Software Cache Implementation for the Cell-BE Processor. In *Proc. of the 20th International Workshop on Languages and Compilers for Parallel Computing, LNCS Vol. 5234*, pages 125–140, Oct. 2007.

[16] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, New York, NY, USA, 2008. ACM.

[17] P. Bellens, J. M. Pérez, R. M. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, page 86, 2006.

[18] F. Blagojevic, X. Feng, K. Cameron, and D. Nikolopoulos. Modeling multigrain parallelism on heterogeneous multi-core processors: A case study of the cell be. In P. StenstrÃűm, M. Dubois, M. Katevenis, R. Gupta, and T. Ungerer, editors, *High Performance Embedded Architectures and Compilers*, volume 4917 of *Lecture Notes in Computer Science*, pages 38–52. Springer Berlin / Heidelberg, 2008.

[19] F. Blagojevic, D. S. Nikolopoulos, A. Stamatakis, and C. D. Antonopoulos. Dynamic Multigrain Parallelization on the Cell Broadband Engine. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[20] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPoPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 207–216, New York, NY, USA, 1995. ACM.

[21] J. L. Bosque and L. Pastor. A Parallel Computational Model for Heterogeneous Clusters. *Parallel and Distributed Systems, IEEE Transactions on*, 17(12):1390–1400, dec. 2006.

[22] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM '99: Proceedings of the 18th IEEE Conference on Computer Communications*.

[23] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine and Its First Implementation—A Performance View. *IBM Journal of Research and Development*, Sept. 2007.

[24] T. Chen, Z. Sura, K. M. O'Brien, and J. K. O'Brien. Optimizing the Use of Static Buffers for DMA on a CELL Chip. In *Languages and Compilers for Parallel Computing, 19th International Workshop (LCPC)*, pages 314–329, 2006.

[25] T. Chen, T. Zhang, Z. Sura, and M. G. Tallada. Prefetching irregular references for software cache on cell. In *CGO '08: Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 155–164, New York, NY, USA, 2008. ACM.

[26] J. P. Corrigna. Letter to Market Data Recipients. http://opradata.com/specs/Traffic_Projections_2009_2010.pdf.

[27] C. H. Crawford, P. Henning, M. Kistler, and C. Wright. Accelerating Computing With the Cell Broadband Engine Processor. In *Proceedings of the 2008 ACM Conference on Computing Frontiers (CF08)*, pages 3–12, 2008.

[28] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *PPoPP '93: Proceedings of the 4th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 1–12, New York, NY, USA, 1993. ACM.

[29] W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J. H. Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with Streams. In *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing (Supercomputing'2003)*, page 35, 2003.

[30] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil Computation Optimization and Auto-Tuning on State-of-the-Art Multicore Architectures. In *SC '08: Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*.

[31] A. Duran, J. M. Perez, E. Ayguade, R. M. Badia, and J. Labarta. Extending the OpenMP Tasking Model to Allow Dependent Tasks. In *OpenMP in a New Era of*

Parallelism, Proceedings of the 4th International Workshop on OpenMP, LNCS Vol. 5004, pages 111–122, July 2008.

[32] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[33] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, and M. Gschwind. Optimizing compiler for the cell processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[34] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, page 83, 2006.

[35] X. Feng, K. W. Cameron, and D. A. Buell. PBPI: A High Performance Implementation of Bayesian Phylogenetic Inference. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, page 75, 2006.

[36] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[37] A. Funk, V. Basili, L. Hochstein, and J. Kepner. Application of a development time productivity metric to parallel software development. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 8–12, New York, NY, USA, 2005. ACM.

[38] B. Gedik, H. Andrade, K.-L. Wu, P. S. Yu, and M. Doo. SPADE: The System S Declarative Stream Processing Engine. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*.

[39] M. I. Gordon, W. Thies, and S. P. Amarasinghe. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 151–162, 2006.

[40] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.

[41] D. Grossman. The Transactional Memory / Garbage Collection Analogy. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 695–706, New York, NY, USA, 2007. ACM.

[42] J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: Programming General-Purpose Multicore Processors Using Streams. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 297–307, 2008.

[43] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple instruction stream processor. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 114–127, Washington, DC, USA, 2006. IEEE Computer Society.

[44] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[45] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41:33–38, July 2008.

[46] M. Hirzel, H. Andrade, B. Gedik, V. Kumar, G. Losa, M. Mendell, H. Nasgaard, R. Soule, and K.-L. Wu. Streams Processing Language Specification. Technical Report RC24897, IBM Research, 2009.

[47] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978.

[48] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili. Parallel programmer productivity: A case study of novice parallel programmers. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC '05, pages 35–, Washington, DC, USA, 2005. IEEE Computer Society.

[49] S. S. Huang, A. Hormati, D. F. Bacon, and R. Rabbah. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 76–103, Berlin, Heidelberg, 2008. Springer-Verlag.

[50] W. Hundsdorfer. Numerical Solution of Advection-Diffusion-Reaction Equations. Technical report, Centrum voor Wiskunde en Informatica, 1996.

[51] IBM. *Accelerated Library Framework Programmer's Guide and API Reference*, 2007.

[52] IBM. *Software Development Kit for Multicore Acceleration*, 2007.

[53] IBM, Sony, Toshiba. *Cell Broadband Engine Programming Handbook*, 2007.

[54] IEEE. IEEE Standard for Information Technology- Portable Operating System Interface (POSIX) Base Specifications, Issue 7, Volume 2.

[55] Intel. Intel C++ Compiler User and Reference Guides. Intel Document number: 304968-022US, 2008.

[56] G. Jacques-Silva, J. Challenger, L. Degenaro, J. Giles, and R. Wagle. Towards Autonomic Fault Recovery in System-S. In *ICAC '07: Proceedings of the Fourth IEEE International Conference on Autonomic Computing*.

[57] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *SIGMOD '06: The 2006 ACM SIGMOD International Conference on Management of Data*.

[58] J. Järvi and J. Freeman. Lambda Functions for C++0x. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 178–183, New York, NY, USA, 2008. ACM.

[59] J. Järvi and J. Freeman. C++ Lambda Expressions and Closures. *Science of Computer Programming*, 75:762–772, September 2010.

[60] L. V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 91–108, New York, NY, USA, 1993. ACM.

[61] Khronos OpenCL Working Group. The OpenCL Specification. 2008.

[62] T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 226–236, New York, NY, USA, 2007. ACM.

[63] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81 – 92, 2003.

[64] D. Kunzman, G. Zheng, E. Bohm, and L. V. KalÃľ. Charm++, Offload API, and the Cell Processor. In *Proceedings of the Workshop on Programming Models for Ubiquitous Parallelism*, 2006.

[65] D. M. Kunzman and L. V. Kalé. Towards a Framework for Abstracting Accelerators in Parallel Applications: Experience with Cell. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, USA, 2009. ACM.

[66] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 63–74, New York, NY, USA, 1991. ACM.

[67] S.-w. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou. Machine Learning-Based Prefetch Optimization for Data Center Applications. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*.

[68] A. W. Lim, G. I. Cheong, and M. S. Lam. An affine partitioning algorithm to maximize parallelism and minimize communication. In *ICS '99: Proceedings of the 13th international conference on Supercomputing*, pages 228–237, New York, NY, USA, 1999. ACM.

[69] A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 103–112, New York, NY, USA, 2001. ACM.

[70] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu. Multi-core Acceleration of Chemical Kinetics for Simulation and Prediction. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*.

[71] J. C. Linford and A. Sandu. Optimizing Large Scale Chemical Transport Models for Multicore Platforms. In *Proceedings of the 2008 Spring Simulation Multiconference*, Ottawa, Canada, April 14–18 2008.

[72] T. Liu, H. Lin, T. Chen, J. K. O'Brien, and L. Shao. DBDB: Optimizing DMA Transfer for the Cell BE Architecture. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 36–45, New York, NY, USA, 2009. ACM.

[73] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning From Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[74] X. Martorell, E. Ayguade, N. Navarro, J. Corbalan, M. Gonzalez, and J. Labarta. Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors. In *ICS '99: Proceedings of the 13th International Conference on Supercomputing*.

[75] T. Mattson. Introduction to OpenMP—Tutorial. In *SC '06: Proceedings of the Conference on High Performance Computing, Networking, Storage and Analysis*.

[76] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM*, 3:184–195, April 1960.

[77] M. D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In *Proceedings of GSPx Multicore Applications Conference*, 2006.

[78] M. D. McCool and B. D'Amora. Programming using RapidMind on the Cell BE. In *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing (Supercomputing'2006)*, page 222, 2006.

[79] R. McIlroy, P. Dickman, and J. Sventek. Efficient dynamic heap allocation of scratch-pad memory. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*, pages 31–40, New York, NY, USA, 2008. ACM.

[80] R. McIlroy and J. Sventek. Hera-JVM: Abstracting Processor Heterogeneity Behind a Virtual Machine. In *Proc. of the 12th Workshop on Hot Topics in Operating Systems (HotOS 2009)*, May 2009.

[81] R. McIlroy and J. Sventek. Hera-jvm: a runtime system for heterogeneous multi-core architectures. In *OOPSLA '10: Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 205–222, New York, NY, USA, 2010. ACM.

[82] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 2.2.

[83] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Proceedings of Eighth IEEE Symposium on Software Metrics, 2002.*, pages 87–94, 2002.

[84] New York Times. Intel Acquires RapidMind, a Software Company. http://dealbook.blogs.nytimes.com/2009/08/24/intel-acquires-software-company-rapidmind, August 24, 2009.

[85] D. Nuzman, I. Rosen, and A. Zaks. Auto-vectorization of interleaved data for simd. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 132–143, New York, NY, USA, 2006. ACM.

[86] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting openmp on cell. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 65–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[87] OpenMP Architecture Review Board. OpenMP Application Program Interface, v. 3.0, May 2008. http://www.openmp.org/mp-documents/spec30.pdf.

[88] B. Rose. CellStream. http://www.cs.vt.edu/~bar234/cellstream.

[89] A. Sandu, D. Daescu, G. Carmichael, and T. Chai. Adjoint Sensitivity Analysis of Regional Air Quality Models. *Journal of Computational Physics*, 204:222–252, 2005.

[90] S. Schneider, H. Andrade, B. Gedik, K.-L. Wu, and D. S. Nikolopoulos. Evaluation of Streaming Aggregation on Parallel Hardware Architectures. In *DEBS '10: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*.

[91] S. Schneider, J.-S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos. A Comparison of Programming Models for Multiprocessors with Explicitly Managed Memory Hierarchies. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[92] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[93] L. Strozek and D. Brooks. Energy- and area-efficient architectures through application clustering and architectural heterogeneity. *ACM Trans. Archit. Code Optim.*, 6:4:1–4:31, April 2009.

[94] A. S. van Amesfoort, A. L. Varbanescu, H. J. Sips, and R. V. van Nieuwpoort. Evaluating Multi-Core Platforms for HPC Data-Intensive Kernels. In *CF '09: Proceedings of the 6th ACM Conference on Computing Frontiers*.

[95] N. Vujic, M. Tallada, X. Martorell, and E. Ayguade. Automatic Prefetch and Modulo Scheduling Transformations for the Cell BE Architecture. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):494 –505, apr. 2010.

[96] P. H. Wang, J. D. Collins, G. N. Chinya, H. Jiang, X. Tian, M. Girkar, N. Y. Yang, G.-Y. Lueh, and H. Wang. EXOCHI: Architecture and Programming Environment for a Heterogeneous Multi-core Multithreaded System. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 156–166, 2007.

[97] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, and K.-L. Wu. SODA: An Optimizing Scheduler for Large-Scale Stream-Based Distributed Computer Systems. In *Middleware '08: Proceedings of the 9th International Middleware Conference*.

[98] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM.

[99] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang. Pangaea: a tightly-coupled ia32 heterogeneous chip multiprocessor. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 52–61, New York, NY, USA, 2008. ACM.

[100] K.-L. Wu, P. S. Yu, B. Gedik, K. W. Hildrum, C. C. Aggarwal, E. Bouillet, W. Fan, D. A. George, X. Gu, G. Luo, and H. Wang. Challenges and Experience in Prototyping a Multi-Modal Stream Analytic and Monitoring Application on System S. In *VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases*.

[101] Xbit Laboratories. IBM Plans to Integrate Cell Chip into "Future Power Roadmap". http://www.xbitlabs.com/news/cpu/display/20101008162445_IBM_Plans_to_Integrate_Cell_Chip_into_Future_Power_Roadmap.html, October 8, 2010.

[102] L. Xue, M. Kandemir, G. Chen, F. Li, O. Ozturk, R. Ramanarayanan, and B. Vaidyanathan. Locality-Aware Distributed Loop Scheduling for Chip Multiprocessors. In *20th International Conference on VLSI Design*, pages 251–258, Jan. 2007.

[103] L. Xue, M. Kandemir, G. Chen, and T. Yemliha. SPM Conscious Loop Scheduling for Embedded Chip Multiprocessors. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 391–400, Washington, DC, USA, 2006. IEEE Computer Society.

[104] J.-S. Yeom and D. S. Nikolopoulos. Strider: Runtime support for optimizing strided data accesses on multi-cores with explicitly managed memories. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[105] X. J. Zhang, H. Andrade, B. Gedik, R. King, J. Morar, S. Nathan, Y. Park, R. Pavuluri, E. Pring, R. Schnier, P. Selo, M. Spicer, and C. Venkatramani. Implementing a High-Volume, Low-Latency Market Data Processing System on Commodity Hardware using IBM Middleware. In *WHPCF '09: Workshop on High Performance Computational Finance*.

Appendix A

Compiler Design

Exploring compiler designs was not the purpose of this work. However, in order for the project to succeed, we needed a source-to-source compiler with a code base small enough for a single developer to handle, yet still be extensible enough for us to experiment with new features. As such, considerable time and effort did go into the software architecture for Cellgen. We present the design in this appendix.

Cellgen depends on Boost.Spirit [2] for parsing. Through template metaprogramming and operator overloading, the Spirit framework allows language grammars to be written directly in C++. The Spirit framework returns parse trees to the calling program. Note that since Cellgen is a source-to-source compiler, it operates on parse trees, not abstract syntax trees—it needs to reproduce syntactic constructions such as braces that are removed from abstract syntax trees. After parsing, Cellgen performs semantic analysis and code generation phases, which are the novel aspects of its design. We will explain how these phases work and the main data structures that these phases operate on.

A.1 Files

Cellgen accepts a single source file which must have a .cellgen extension. A Cellgen source file can have multiple accelerated regions. From this source file, Cellgen produces two C files: one for the PPE, and one for the SPE. These two source files are compiled using the Make files provided by IBM's SDK. The runtime system code is copied into the directory which contains the Cellgen source file.

A.2 Phases of Compilation

The initial phase of compilation is the parsing of the Cellgen file itself. We use the Boost.Spirit library for parsing. The parsing phase constructs a parse tree for the offloaded code, and a data structure which captures all of the pertinent information that was a part of the Cellgen directive itself: the set of private variables, the set of shared variables, the set of reductions, the reduction operator, and a reference into the parse tree. During semantic analysis, this reference is used to pair an accelerated region with where the actual code for that region begins.

After the parse tree has been created and a data structure exists for each accelerated region, the semantic analysis can begin. The semantic analysis is where the transformations from Chapter 3 take place: reference analysis, access analysis, buffer substitution and stencil support. Some of these analyses impact other transformations. For example, the access analysis determines if a shared variable is an *in*, out *out* or *inout* variable. The transformation objects for buffer substitutions need to know what kind of variable a shared variable is in order to generate the correct data transfers. Similarly, the kind of access (contiguous or strided) and if stencil support is needed changes the behavior of the various transformations.

The semantic analysis proceeds in a similar manner as a recursive-descent parse. As the analysis recurses through the parse tree, it uses the combined knowledge of the particular code it sees at a node, the code seen in previous nodes, and knowing where in the grammar it is to infer programmer intent and what kind of code to generate.

A.3 Transformation Objects

Clearly, Cellgen must modify user code. Most of this modification is in the form of additions (such as adding code to handle the data transfers), but it also does need to replace and remove portions of the original code (such as when loops are duplicated for remainder operations, but only some of the loop must execute). However, modifying the parse tree directly is potentially messy—it leaves no record of what was changed, which makes it difficult to infer in later stages what semantic actions were taken in the case that action needs to be undone.

We have taken a functional approach to code generation. During semantic analysis, Cellgen adds transformation objects (called xformers in the code) to the relevant nodes of the parse tree. Transformation objects are function objects responsible for generating code that obey the following interface:

```
struct xformer: public unary_function<const string&, string> {
  virtual ~xformer() {}
  virtual xformer* clone() const = 0;
```

```
  virtual string class_name() const = 0;
  virtual string operator()(const string& old) = 0;
};
```

In C++, function objects are objects that implement the application operator, **operator**(). They are objects, so they can store information and be passed around a program easily, but they can also be used as functions. In C++, function objects are used in places where functional languages would use closures [59, 58].

In Cellgen, a transformation object (a xformer) is an object that when used as a function takes a string as an argument, and returns a string as a result. In most cases, the result will be a *transformation* of the original string. Consequently, transformation objects can be composed; the result from one can be the input to another. This ability is needed when multiple transformation objects exist on a single node. The following is the transformation object used to call the function which determines the iteration bounds for each SPE:

```
class compute_bounds: public xformer {
  const string least;
public:
  compute_bounds(const string& l): least(l) {}
  string operator()(const string& old)
  {
    return old + "compute_bounds(&spe_start, &spe_stop, sizeof(" + least + "));";
  }

  xformer* clone() const { return new compute_bounds(*this); }
  string class_name() const { return "compute_bounds"; }
};
```

In order to explain how transformation objects are used, we must present the parse tree that Cellgen operates on.

A.4 Augmented Parse Trees

Spirit is parameterized on what kinds of nodes its parse trees contain. We extended the default node (through inheritance) to also contain a list of transformation objects. Adding a transformation list to each node means that nodes in the parse tree always carry with them all of the information needed to generate code at that node. This makes the code generation phase simple: if the node has no transformation objects, then output the text contained at that node. If the node has transformation objects, compose them with each other, passing

the original text into the first one. (This operation is an *accumulation*, or a *right fold* where the list itself contains the functions to be applied.)

Note that the transformation objects are created during semantic analysis, but they are not executed until code generation. Using function objects, whose execution can be deferred until code generation time, enables this technique. By creating function objects whose execution is deferred, we also have the flexibility to add or remove transformations in later stages of semantic analysis.

A.5 Example

To clarify the phases of compilation, we will walk through a simple example. The following code implements the sum over an array of integers:

```
#pragma cell shared(int* n = nums) private(int N = N) reduction(+: int s = sum)
{
  int i;

  for (i = 0; i < N; ++i) {
    s += n[i];
  }
}
```

Figure A.1 is the augmented parse tree for this code. All of the terminal and non-terminals are represented with normal text, and the gray dashed lines show the structure of the tree. Some of the nodes in the tree have lists of transformation objects, which are represented with bold text and red arrows. Spirit produces the parse tree, which Cellgen traverses during semantic analysis. During this phase, Cellgen adds transformation objects to generate code based on reference analysis, access analysis, buffer substitution, choosing the buffer size, and other infrastructure requirements.

While the original text of a node is never directly changed, the tree itself is changed in two circumstances. First, the transformation that converts a shared variable access to a buffer access subsumes all of the nodes in the tree below it. Consequently, Cellgen removes those nodes from the tree. Second, Cellgen duplicates the main computation of the tree. Cellgen transforms the first instance of the main computation to operate on buffer-size multiples. The second instance operates on the remaining data.

Computation duplication is possible because of the transformers. The transformer objects are duplicated along with the rest of the sub-tree, but like Hox genes that inform a cell of its location in an organism during embryonic development, transformers are told where in the parse tree they are. Some transformers generate different code depending on their location.

If Cellgen modified the text in the parse tree directly, duplicating the tree and getting the correct behavior would be significantly more difficult.

Figure A.1: Augmented parse tree of the example code.

Applying the transformation objects in order at each node results in the following SPE code:

```
// define_var(prev)
int prev = 0;

// define_var(___i___)
int ___i___ = 0;

// compute_bounds
compute_bounds(&SPE_start, &SPE_stop, sizeof(int));
```

```
// def_clipped_range
int __N__ = min((SPE_stop − SPE_start), 16384 / sizeof(int));

// max_buffer_size
const int n_buf_sz = (__N__ / 3) − ((__N__ / 3) % 16);

// buffer_allocation
int *n_buf = _malloc_align(sizeof(int) * 2 * n_buf_sz, 7);

// def_buffer
int *n;
n = n_buf;

// def_next
int n_nxt = 0;

// def_rem
int n_rem;

// def_ful
int n_ful;

// def_reduction
int s = *s_red;

int i;

// get_in_first<row_access>
DMA_get(n_buf + n_buf_sz * n_nxt, (unsigned long) (n_adr + (SPE_start)),
        sizeof(int) * n_buf_sz, n_nxt);

// reset_rem
n_rem = ((SPE_stop − SPE_start) % (n_buf_sz));

// reset_ful
n_ful = SPE_stop − n_rem;

// variable_name, variable_name, naked_string, loop_increment
for (i = SPE_start; i < n_ful; i += n_buf_sz) {
  // gen_in<row_access>
  prev = n_nxt;
  n_nxt = (n_nxt + 1) % 2;
  DMA_wait(n_nxt, fn_id);
  DMA_get(n_buf + n_buf_sz * n_nxt, (unsigned long) (n_adr + i + n_buf_sz),
```

```
            sizeof(int) * (i + n_buf_sz < n_ful ? n_buf_sz : n_rem), n_nxt);
    n = n_buf + n_buf_sz * prev;
    DMA_wait(prev, fn_id);

    // buffer_loop_start, buffer_loop_stop
    for (__i__ = 0; __i__ < n_buf_sz; ++__i__) {
      // to_buffer_space
      s += n[__i__];
    }
  }
  if (n_rem) {
    // gen_in_row<access>
    n = n_buf + n_buf_sz * n_nxt;
    DMA_wait(n_nxt, fn_id);

    // buffer_loop_start, buffer_loop_stop
    for (__i__ = 0; __i__ < n_rem; ++__i__) {
      // to_buffer_space
      s += n[__i__];
    }
  }

  // buffer_deallocation
  _free_align(n_buf);

  // reduction_assign
  *s_red = s;
```

A.6 Runtime System

Much of the generated code are calls into the runtime system. The runtime system is a heavily modified version of the runtime system initially presented by Blagojevic et al. [19] and an initial version of CellStrider presented by Yeom et al. [104]. This runtime system allows for a looser coupling between the compiler and the actual code executed at runtime. For example, Cellgen will generate calls to `dma_get`. If there is an issue with the implementation of DMAing memory into the SPEs, it can be fixed without modifying the compiler source itself.