Evaluation of Streaming Aggregation on Parallel Hardware Architectures

Scott Schneider[†,♠] Henrique Andrade[♠] Buğra Gedik[♠] Kun-Lung Wu[♠]
Dimitrios S. Nikolopoulos[‡]

[†] Virginia Tech
Department of Computer Science
Blacksburg, VA, USA
scschnei@cs.vt.edu

[♠] T. J. Watson Research Center
IBM Research
Hawthorne, NY, USA
{hcma,bgedik,klwu}@us.ibm.com

[‡] Institute for Computer Science
Foundation for Research and Technology–Hellas
Heraklion, Greece
dsn@ics.forth.gr

ABSTRACT

We present a case study parallelizing streaming aggregation on three different parallel hardware architectures. Aggregation is a performance-critical operation for data summarization in stream computing, and is commonly found in sense-and-respond applications. Currently available commodity parallel hardware provides promise as accelerators for streaming aggregation. However, how streaming aggregation can map to the different parallel architectures is still an open question. Streaming aggregation is obviously data parallel, but in practice its performance relies more on efficient data movement than computation, as we will demonstrate. Furthermore, we used workloads such as stock market data, which introduces unique data distribution problems. The three parallel architectures we use in our study are an Intel Core 2 Quad processor, an Nvidia GTX 285 GPU and the IBM PowerXCell 8i, an enhanced version of the Cell Broadband Engine architecture. Our implementations use OpenMP, CUDA and Cellgen (a compiler for OpenMP-like support on Cell) respectively. We find that the Cell's programmable local storage, and its low latency, high bandwidth access to main memory are best suited for parallelizing streaming aggregation. GPUs in the future can overcome the latency and bandwidth limitations by being fully integrated in the system's memory hierarchy. In order to attain good performance on existing parallel architectures, we find that developers must characterize their problem in terms of communication versus computation costs; memory access patterns, including assessing whether their algorithms reuse data; and the granularity of data access patterns.

1. INTRODUCTION

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 [17] 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 [2], which is a homogeneous multicore similar in principle to an SMP. On the other end we have an Nvidia GeForce GTX 285 GPU [4], 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 [9], which is better suited at data parallel computations than a homogeneous multicore [7] 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 [26]. 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 [6, 26]. The frequency of trades for a particular symbol roughly follow Zipf's law [8], 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. [11] used a code generator specific to stencil computations, and the work of Linford et al. [19] 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. [23] 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 have to implement and deploy large scale streaming systems.

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.

2.1 Streaming Aggregation

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 [5, 15, 16, 24, 25] and the SPADE (Stream Processing Application Declarative Engine) programming language [12, 13] 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.

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* 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* [6]. 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.

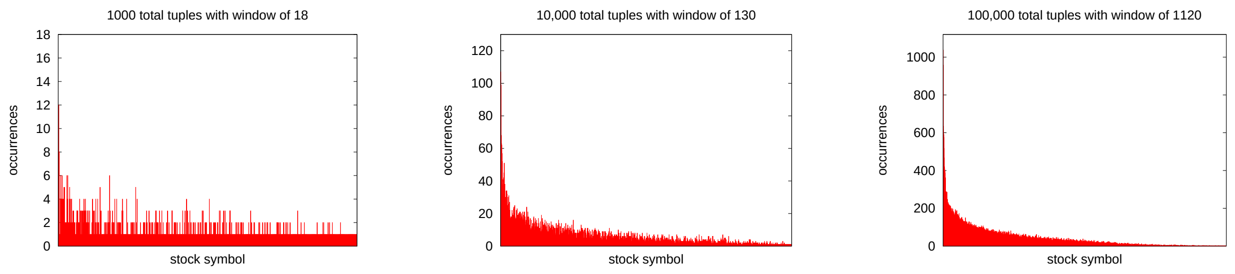


Figure 1: Stock symbol frequency distribution histograms.

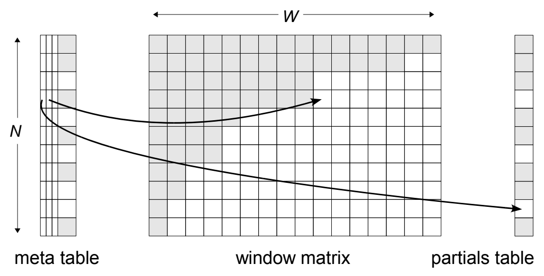


Figure 2: Data structures used in partial aggregations.

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.

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 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.

2.3 Stock Market Distribution

Our data is a set of stock 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 [8].

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 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.

3. CASE STUDY

Our case study compares the performance of three implementations of streaming aggregation on three different parallel archi-

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

Table 1: Hardware.

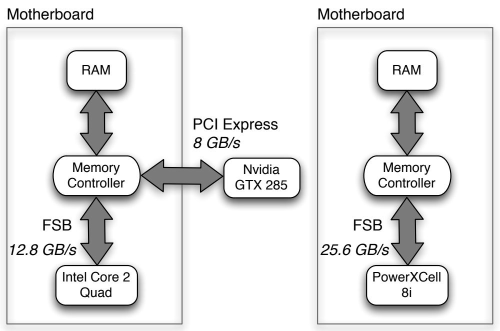


Figure 3: Physical layout of our experimental machines. The node with the Intel Quad and Nvidia GPU are on the left, and the Cell node is on the right.

tectures. The parallel architectures represent the current spectrum of multicore processors available to developers. All three implementations 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.

3.1 Parallel Hardware

We list the specifications for the hardware used in our study in Table 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 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 [18]. 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.

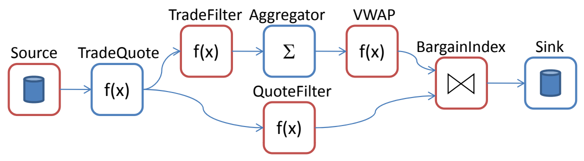


Figure 4: Stock market bargain discovery using SPADE. Our work occurs in the *Aggregator* operator.

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 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. A Cell processor has a Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). The PPE is an in-order PowerPC core with hardware controlled caches. The SPEs are in-order, SIMD cores with a 256 KB local storage. The local storage is not hardware controlled; explicit Direct Memory Accesses (DMAs) issued by software are the only way to access main memory.

3.2 Implementations

We derived our synthetic experiments from the SPADE application for stock market bargain discovery depicted in Figure 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 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 5, follow the fork-join model of data parallelism [20]. 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 2.1.

3.2.1 *OpenMP*

OpenMP [21], 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 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 [14].

3.2.2 *CUDA*

CUDA [3] is an architecture for general purpose programming on GPUs which provides language extensions for C. Note that the code in Figure 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 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 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.

3.2.3 *Cellgen*

Cellgen [1, 22] is a directive based parallel language and compiler similar to OpenMP. Based on memory access patterns in high-level code, Cellgen generates Cell-specific data transfers. These transfers stream data from main memory through the SPEs using multi-buffering, which hides data transfer latency by overlapping it with computation. 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.

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 [10], 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 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 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.

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.

```
#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].svwap += table->matrix[g][i].svwap;
    partials[n].svolume += table->matrix[g][i].svolume;
    ++partials[n].count;

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

      partials[n].count = 0;
      partials[n].svwap = 0;
      partials[n].svolume = 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].svwap += matrix[g][i].svwap;
    partials[n].svolume += matrix[g][i].svolume;
    ++partials[n].count;

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

      partials[n].count = 0;
      partials[n].svwap = 0;
      partials[n].svolume = 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 svwap = partials[g].svwap;
    float svolume = partials[g].svolume;
    int count = partials[g].count;

    float res_svwap;
    float res_svolume;
    char send = 0;

    if (next == 0) {
      continue;
    }

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

      if (count == W) {
        res_svwap = svwap;
        res_svolume = svolume;

        send = 1;
        svwap = 0;
        svolume = 0;
        count = 0;
      }
    }

    meta[g].rslt.svwap = res_svwap;
    meta[g].rslt.svolume = res_svolume;
    meta[g].send = send;
    meta[g].next = 0;

    partials[g].svwap = svwap;
    partials[g].svolume = svolume;
    partials[g].count = count;
  }
}
```

Figure 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.

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

Table 2: Data involved in each aggregation.

4.1.1 Sequential

Our sequential implementation runs on a single core on our Intel Quad node (Figure 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 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.

4.1.2 OpenMP on Intel Quad

For the OpenMP implementation on the Intel Quad (Figure 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.

As seen in Figure 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

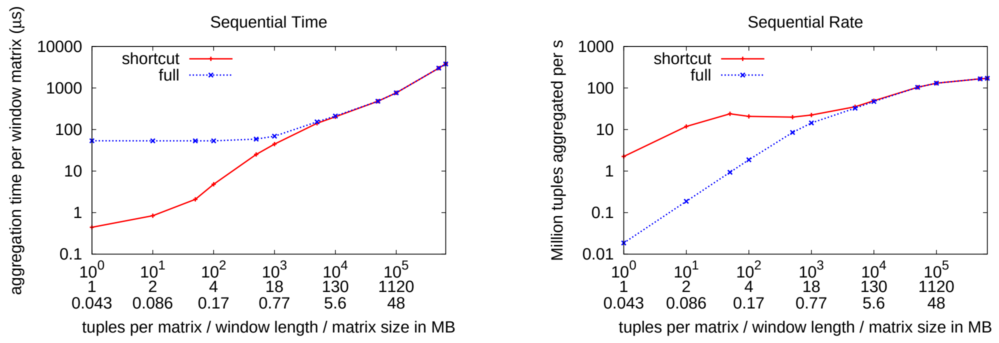


Figure 6: Performance of two sequential implementations.

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

Table 3: Ratio of data transfer time to total aggregation time for GPU and Cell.

cache misses. Since the amount of computation is small compared to data movement time, these cache misses turn into stall cycles.

4.1.3 CUDA on GPU

In Figure 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 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 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.

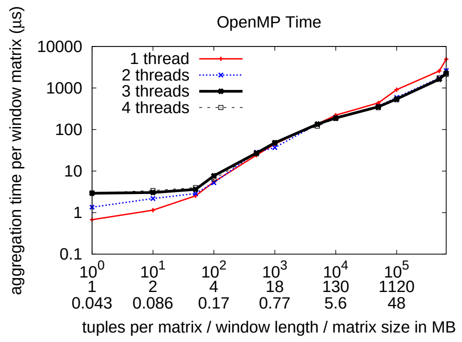
4.1.4 Cellgen on Cell

We varied the number of SPEs used in our Cell implementation and compared that to the performance of the PPE, as seen in Figure 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 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 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.

4.2 Inter-implementation Comparison

We compare all of the implementations in Figure 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



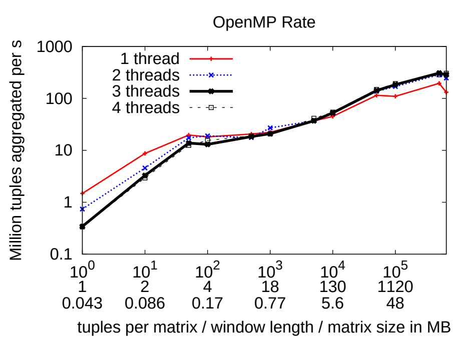
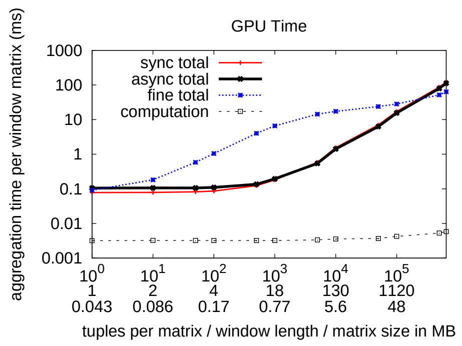


Figure 7: Performance of different number of threads used in the Intel Quad implementation.



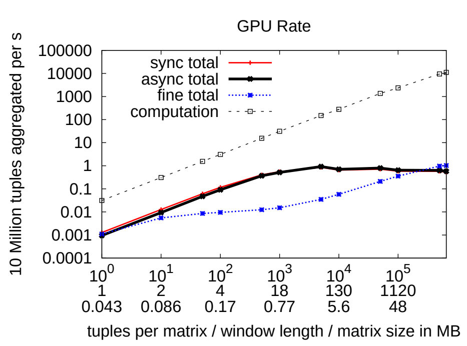


Figure 8: Performance of different data transfer strategies for the GPU implementations.

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 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 coprocessors.

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.

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

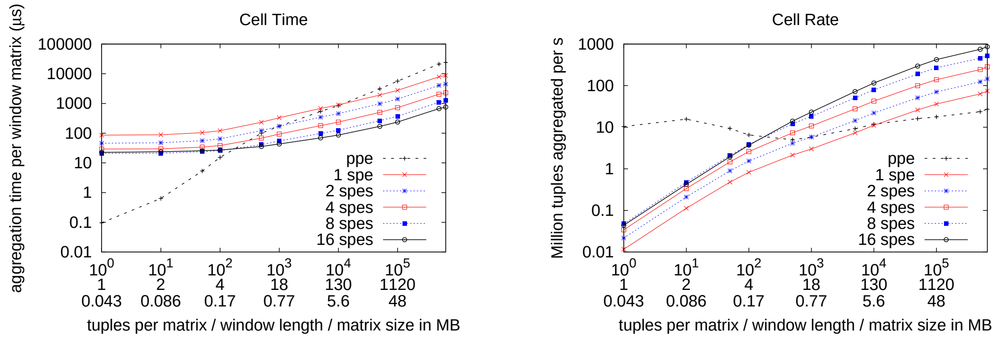


Figure 9: Performance of different number of SPEs used for the Cell implementation.

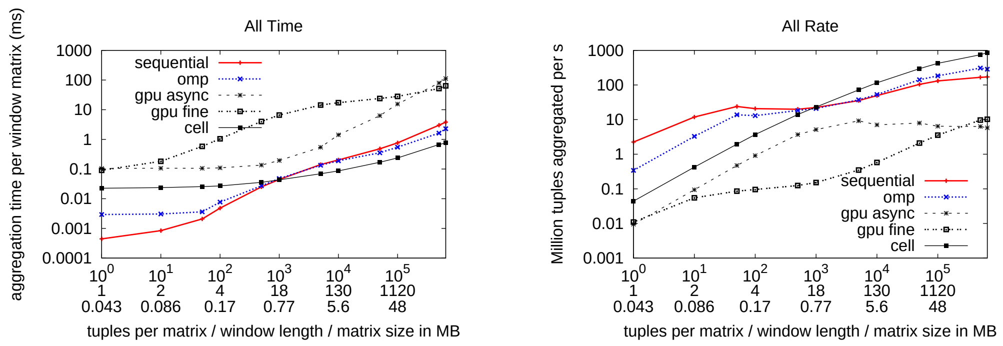


Figure 10: Performance comparison of all implementations.

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 like the Intel Quad and Cell. The computational potential for GPUs is extraordinary, but we are limited by the granularity of its memory transfers.

6. REFERENCES

[1] Cellgen. http://www.cs.vt.edu/~scschnei/cellgen.

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

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

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

[5] 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*.

[6] 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*.

[7] 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*.

[8] 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*.

[9] 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.

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

[11] 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*.

[12] 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*.

[13] 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.

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

[15] 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*.

[16] 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*.

[17] Khronos OpenCL Working Group. The OpenCL Specification, 2009. http://www.khronos.org/opencl.

[18] 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*.

[19] 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*.

[20] 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*.

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

[22] 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*.

[23] 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*.

[24] 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*.

[25] 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*.

[26] 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*.