

# Understanding the Design Trade-offs among Current Multicore Systems for Numerical Computations

Seunghwa Kang David A. Bader Richard Vuduc  
Georgia Institute of Technology  
Atlanta, GA, 30332, USA

## Abstract

*In this paper, we empirically evaluate fundamental design trade-offs among the most recent multicore processors and accelerator technologies. Our primary aim is to aid application designers in better mapping their software to the most suitable architecture, with an additional goal of influencing future computing system design. We specifically examine five architectures, based on: the Intel quad-core Harpertown processor, the AMD quad-core Barcelona processor, the Sony-Toshiba-IBM Cell Broadband Engine processors (both the first-generation chip and the second-generation PowerXCell 8i), and the NVIDIA Tesla C1060 GPU. We illustrate the software implementation process on each platform for a set of widely-used kernels from computational statistics that are simple to reason about; measure and analyze the performance of each implementation; and discuss the impact of different architectural design choices on each implementation.*

## 1 Introduction

Driven by the limits to single-core performance due to power constraints, multicore processors and accelerator architectures are becoming ubiquitous in every computing area. These systems are attractive to application developers because of their impressive peak computational potential and (in several cases) their energy-efficient processing capabilities. However, the architectures themselves are diverse and reflect a wide variety of design trade-offs. Consequently, we might expect the performance of a given application to be an even more sensitive function of the architecture than in previous generation single-core general purpose processors, which in turn is expected to affect software development costs significantly. The longer-term goal of our research effort is to better understand the impact of the architectural design trade-offs on software.

The research literature on software optimization for these multicore systems is growing rapidly, particularly for platforms based on the Cell/B.E. [5, 12, 27] and for GPUs [21, 30]. Ryoo, *et al.* [25, 26] publish extensively on generalizing optimization principles for the NVIDIA GPUs based on the CUDA framework [19]. In addition to the optimization research for a single platform, several inter-architecture comparisons are published as well. Williams, *et al.* [31] compare the performance of emerging multicore platforms, including the AMD dual-core Opteron processor, the Intel quad-core Harpertown processor, the SUN Niagara processor, and the Cell/B.E. processor for sparse matrix-vector multiplication. Also, there are a few studies that compare the performance of CPUs, GPUs, FPGAs, the Cell/B.E. processor, and the Cray MTA-2 [16, 7, 5].

The focus of this paper is on evaluating the impact of fundamental design trade-offs on a particular class of widely-used but simple-to-analyze software kernels. We study a range of systems with native double-precision support, described in detail in Section 2, including a two processor (2P) system with the Intel quad-core Harpertown 2.5 GHz E5420 processors, a four processor (4P) system based on the AMD quad-core Barcelona 2.0 GHz 8350 processors, an IBM QS20 blade with two Cell/B.E. 3.2 GHz processors, an IBM QS22 blade with two PowerXCell 8i 3.2 GHz processors, and a desktop system equipped with one NVIDIA Tesla C1060 GPU. As the test systems have varying numbers of chips, clock frequencies, prices, and power consumptions, we focus on architectural design trade-offs and their impacts on different kernels rather than identifying the best performing processor.

We evaluate these systems experimentally using three kernels from computational statistics with differing computational characteristics. First, we create two kernels by extracting the most computationally intensive part of the covariance/correlation computation from the R statistics package [3]. These kernels are based on Pearson's method and Kendall's method [13], which we hereafter refer to as Ker-

nel1 and Kernel2, respectively. The third kernel (Kernel3) is created by modifying Kernel2 to highlight each system’s capability in processing highly floating-point intensive computation. Section 3 describes these kernels in more detail.

We also discuss our implementation and software optimization process, to highlight the challenges and complexities of software development for each architecture (Section 4). However, we consider the main contribution of this paper to be our inter-architectural analysis, not the optimization work. Our experimental results highlight the performance of each system in executing different mix of instructions for compute-bound and data-intensive cases (Section 5). We consider both single-precision and double-precision performance for floating-point operations and attempt to characterize the resulting performance in terms of each system’s design choices.

## 2 Inter-architectural Design Trade-offs

This section describes five multicore systems of interest in this study, which are summarized in Table 1. In particular, each subsection considers a particular design dimension, and qualitatively summarizes the differences among the architectures. For additional processor details, we refer interested readers elsewhere [9, 8, 14].

Among “conventional” general purpose multicore microprocessors, we consider the Intel quad-core E5420 Harpertown processor and AMD’s quad-core 8350 Barcelona. The Intel Harpertown and AMD Barcelona processors share a similar micro-architecture, but take distinct approaches to cache hierarchy and memory subsystem design. The Intel Harpertown has 12 MB L2 cache memory, and two cores among four cores in a chip share 6 MB L2 cache. The AMD Barcelona has a dedicated 512 KB L2 cache per core in addition to 2 MB L3 cache shared by all four cores in a chip. Also, the AMD Barcelona supports NUMA (Non-Uniform Memory Access) architecture while the Intel Harpertown is based on UMA (Uniform Memory Access) architecture.

Among the multicore accelerator systems, we consider two generations of the STI Cell/B.E. processor and an NVIDIA Tesla C1060 GPU. The Cell/B.E. processor is a heterogeneous multicore processor with one conventional PowerPC core (“PPE”) and eight specialized single-instruction multiple-data (SIMD) accelerators (“SPEs”). The Tesla C1060 is the latest GPU from NVIDIA and delivers 933 Gflop/s in single-precision with native 78 Gflop/s double-precision support. The C1060, based on Tesla architecture [14], has 30 SMs (Streaming Multiprocessors), and each SM has 8 SPs (Streaming Processors) for single-precision and one SP for double-precision support. Also, each SM has two SFUs (Special Function Units) for transcendental functions and attribute interpolation.

## 2.1 Requirements for Parallelism

The Intel Harpertown, AMD Barcelona, and Cell/B.E. processors have four to nine cores, and each core supports SIMD instructions for acceleration. To exploit the parallelism in these processors, these chips require coarse-grain parallelism to exploit their multiple cores in addition to SIMD parallelism.

In contrast, the Tesla C1060 has 240 SPs and 60 SFUs for single-precision. In addition, to be detailed in Section 2.4, the Tesla architecture adopts massive hardware-multithreading to hide the DRAM access latency. The Tesla C1060 requires at least several thousand-way parallelism to exploit its architectural features. Also, the Tesla C1060 requires SIMT (Single Instruction Multiple Threads) parallelism, and every thread in a single warp (a group of 32 threads) needs to agree on the execution path to maximize chip utilization.

## 2.2 Computation Units

Each core of the Intel Harpertown or AMD Barcelona processor can retire up to two SIMD floating-point instructions (one SIMD add and one SIMD multiply) [4, 10]; thus each core can deliver 4 double-precision floating-point operations per cycle. Also the Intel and AMD cores can execute integer instructions in parallel with floating-point instructions. Each SPE in the Cell/B.E. processor has even and odd pipelines. The even pipeline can execute floating-point instructions, fixed point arithmetics, and logical and word-granularity shift and rotate instructions. The odd pipeline executes load/store instructions and fixed point byte-granularity shift, rotate mask, and shuffle instructions. Each SPE can retire one SIMD FMA (fused-multiply-and-add) instructions per cycle. Each SP in the Tesla C1060 executes one scalar instruction per cycle, including a single-precision FMA instruction. Each SFU can also execute four single-precision multiply instructions per cycle. One SP for double-precision support can retire one double-precision FMA instruction per cycle.

The sustainable flop-rate is highly affected by the mix of different instruction types in an execution stream and the structure of the computation unit. The Intel and AMD cores have separate multiply and add units instead of a FMA unit and also can run integer instructions in parallel; thus, they can more flexibly execute different combination of integer and floating-point instructions. Still, to fully utilize both floating-point multiply unit and add unit, this chip requires a 1:1 ratio of multiply and add instructions. One SPE of the Cell/B.E. can issue only one floating-point instruction in a single cycle (whether it is FMA or not), so if multiply cannot be fused with add, the achievable peak flop-rate becomes halved. Still, the Cell/B.E. can run several types of

System	2P Harpertown E5420	4P Barcelona 8350	QS20-Cell/B.E.	QS22-PowerXCell 8i	Tesla C1060
Clock	2.5 GHz	2.0 GHz	3.2 GHz	3.2 GHz	1.296 GHz
Num. chips	2	4	2	2	1
Num. cores / chip	4	4	1 PPE + 8 SPEs	1 PPE + 8 SPEs	30 SMs $\times$ (8 single-precision SPs + 1 double-precision SP + 2 SFUs)
DP Gflop/s	80	128	29.2	204.8	78
SP Gflop/s	160	256	409.6	409.6	933
On-chip memory	6 + 6 MB L2 cache per chip	512 KB L2 cache per core and 2 MB shared cache per chip	256 KB local store per SPE	256 KB local store per SPE	16 KB shared memory + 8 KB constant cache + 16 KB texture cache per SM
DRAM type	DDR2	DDR2	Rambus XDR	DDR2	GDDR3
Shared DRAM access	UMA	NUMA	NUMA	NUMA	N/A
Latency hiding	cache + prefetching	cache + prefetching	double (or triple) buffering	double (or triple) buffering	hardware-multithreading
Theoretical peak bandwidth (GB/s)	21.4	42.8	51.2	51.2	102
Measured bandwidth (GB/s)	8.1 (baseline STREAM)	9.9 (baseline STREAM)	N/A	N/A	73.7 (NVIDIA SDK bandwidthTest)
Power (W)	2 $\times$ 80 (per chip)	4 $\times$ 75 (per chip)	315 (per blade)	250 (per blade)	160 (typical, per board), 200 (max., per board)
Compiler	Intel icc (10.1.018)	Intel icc (10.1.015)	IBM xlc (10.1)	IBM xlc (10.1)	NVIDIA nvcc (release 2.0 V0.2.1221)
Optimization flag	-fast	-fast	-O5 -qarch=cellspu -qtune=cellspu	-O5 -qarch=edp -qtune=edp	-O3 -arch sm_13

Table 1: Summary of the test systems.

fixed point instructions in parallel with floating-point operations. Fully exploiting the SMs in the Tesla C1060 requires FMA instructions and additional multiply instructions for the SFUs. Also, one SP can execute only one instruction per cycle.

### 2.3 Start-up Overhead

Thread creation or kernel launching is faster on the Intel or AMD processors than on the Cell/B.E. or the Tesla C1060. Thus, for a small amount of computation, these general purpose processors outperform the accelerators while the accelerator architectures often exhibit impressive performance for larger data [16].

The test systems also incur different levels of data off-loading overhead. In the Harpertown, Barcelona, and Cell/B.E. processors, the off-loading overhead is largely determined by the off-chip memory bandwidth. In contrast, GPU systems incur additional host memory to the on-board device memory data transfer via a slower PCI Express bus. While GPUs can partially hide the off-loading overhead with asynchronous data transfer (*i.e.*, double-buffering), this

mechanism currently works only for page-locked memory and incurs additional programming overhead [20]. To amortize the off-loading overhead, GPUs require higher computational intensity than other processors [6, 28, 16]. However, the Tesla C1060's on-board memory is much larger (4 GB) than the Harpertown or Barcelona's cache memory (12 or 2 MB) or the Cell/B.E.'s local store (256 KB per SPE  $\times$  8 SPEs). Accordingly, the Tesla C1060 can fit larger data into its on-board memory to minimize the data transfer over the PCI Express bus.

### 2.4 Memory Latency Hiding

The Intel Harpertown and AMD Barcelona processors hide memory latency via cache memory and prefetching mechanisms. The Cell/B.E. overlaps computation with communication via double- or triple-buffering. Double buffering efficiently hides the latency but requires explicit software intervention. The Tesla C1060 tolerates several hundred cycle DRAM access latency via massive hardware-multithreading. The Tesla C1060 also has per SM shared memory (8 KB), constant cache (8 KB), and texture cache

(16 KB). Yet, as each SM can run hundreds of threads in parallel, these on-chip memories have minimum performance impact if there is only a low degree of data sharing among different threads.

## 2.5 Control over On-chip Memory

For the cache-based multicore processors, cache memory is managed by hardware using a LRU (Least Recently Used) policy (or its variants), and programmers have essentially no control over cache partitioning.<sup>1</sup> By contrast, programmers can explicitly manage on-chip (“local store”) memory on the Cell/B.E. The Tesla C1060, along with the NVIDIA CUDA framework, also allows programmers to control the placement of data arrays to the chip’s different types of memories.

## 2.6 Main Memory Access Mechanisms and Bandwidth Utilization

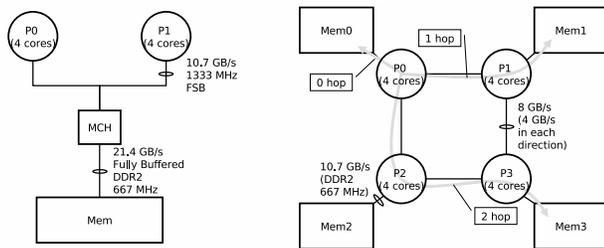


Figure 1: The 2P Intel Harpertown system with UMA architecture (left) and the 4P AMD Barcelona system with NUMA architecture (right).

The AMD Barcelona and Cell/B.E. processors use a NUMA (Non-Uniform Memory Access) architecture, while the Intel Harpertown processor adopts a UMA (Uniform Memory Access) architecture. UMA is conceptually simpler but NUMA has scalability advantages if cores running on different chips access distinct data arrays. In particular, by locating data to the chip’s local main memory, we can minimize the contention and interference in the main memory interface. For instance, if a computation accesses read-only data multiple times, we can replicate the data to each processor’s local DRAM to maximize the bandwidth for accessing the data.<sup>2</sup> The Tesla C1060’s device memory does not support shared memory access over two or more GPUs.

<sup>1</sup>Intel also announced that their upcoming Larrabee processor will better support streaming access by providing a mechanism to mark streaming cache lines for early eviction [29].

<sup>2</sup>Intel has announced a shift to a NUMA approach starting with the future Nehalem processor.

The systems also differ in their deliverable memory bandwidth [22]. In terms of peak aggregate bandwidth, the 2P Harpertown system can deliver 21.4 GB/s, the 4P Barcelona system supports 42.8 GB/s, the QS20 and QS22 blades support 51.2 GB/s for main memory access. The Tesla C1060 supports 102 GB/s peak bandwidth to its 4 GB device memory. However, there is often a significant gap between the peak bandwidth and the sustainable bandwidth [11, 18]. The gap is even larger for the multicore processors due to the interference among multiple threads performing data accesses [17, 23, 24].

The systems adopt different memory controller architectures. In general, most memory controllers are designed to deliver the highest bandwidth when accessing a large contiguous chunk of data, in particular by exploiting maximum locality of a row buffer and bank level parallelism. Switching between DRAM read and write should also be minimized to achieve the highest bandwidth utilization. However, even for simple computations in which each thread is reading a linear array with stride one, memory requests coming from multiple cores can be intermixed, thereby destroying the locality and parallelism of a DRAM chip. For the Intel Harpertown and AMD Barcelona processors, the granularity of memory access is the lowest level cache line size (64 byte). For data-intensive applications, memory access requests from multiple cores with a size of 64 bytes can be heavily interleaved. The situation is even worse for the Intel Harpertown processor with its UMA configuration, as the memory controller hub must mix memory access requests coming out from two different chips. For NVIDIA GPUs, the access granularity is 32, 64, or 128 bytes [20].

By contrast, the Cell/B.E. adopts a different memory subsystem. First, each SPE generates DMA requests with a significantly larger size of up to 16 KB. Even for the data-intensive applications, each SPE issues DMA requests in an intermittent fashion. This minimizes the inter-core interference in DRAM accesses. Therefore, programmers can maximize the bandwidth utilization by increasing the size of DMA accesses [12, 27]. Thus, the Cell/B.E. architecture fundamentally lends itself to higher bandwidth utilization than other systems, in spite of the significant effort toward increasing bandwidth utilization in general purpose multicore processors [17, 15, 11]. The AMD Barcelona processor claims to adopt optimized scheduling algorithms especially for interleaved DRAM access streams as well [4]. Williams, *et al.*, also demonstrate the first-generation Cell/B.E.’s high bandwidth utilization [31], though an open question is what the ultimate impact of adopting different DRAM technologies will be (*i.e.*, first- and second-generation Cell/B.E. architectures adopt different DRAM technologies, namely, XDR and DDR2, respectively). Finally, the Cell/B.E. has an additional advantage in optimizing memory controller, as this processor targets streaming

applications that are highly latency tolerant. The Cell/B.E.’s memory controller can focus on bandwidth utilization, while general purpose multicore processors attempt to address the significantly more difficult problem of balancing bandwidth utilization with fairness and latency issues [18, 17, 23].

## 2.7 Ideal Software Implementations

To optimize the code for the Intel Harpertown, AMD Barcelona, and Cell/B.E. processors, one first needs to identify coarse-grain parallelism and partition data to exploit all the cores. One then needs to consider data layout for higher data transfer and vectorization efficiency and vectorization to exploit SIMD units. The Harpertown and Barcelona processors are significantly less sensitive to data alignment than the Cell/B.E., since they support multiple additional instructions for unaligned data accesses; however, data layout still affects the performance in a non-negligible amount. At high level, optimizing for the Cell/B.E. does not differ much from the Harpertown and Barcelona processors, but the actual implementation is significantly more complex as programmers need to explicitly program for data transfer within the local store size limit of 256 KB. In addition, the gap between the performance of baseline and optimized code is significantly higher for the Cell/B.E., and this often requires manual optimization.

The optimization process for the Tesla C1060 is largely different from the above three processors. For the Tesla C1060, easily identifiable coarse-grain parallelism does not suffice to fully exploit the chip. Thus, the optimization should focus on extracting additional levels of parallelism. To benefit from the high bandwidth and low latency on-chip memories, programmers need to modify an algorithm to maximize data sharing among multiple cores. Data coalescing and broadcasting mechanisms are also crucial for the high performance, and this also needs to be considered in algorithm design. For the Tesla C1060 or other CUDA enabled GPUs, the key challenge arises from high level algorithm design, and the actual implementation is less complex in terms of code size.

For the NUMA-based systems, one can gain significant speedup for bandwidth intensive algorithms by controlling thread binding and data allocation. The optimization result for the Cell/B.E. often is more predictable than the x86 based architectures or the Tesla C1060 owing to its simpler architecture. For the x86 based architectures, the multi-level memory hierarchy with different latency, size, and associativity in each level and complex and adaptive prefetching mechanisms across the memory hierarchy significantly complicate the performance analysis. The Tesla C1060 optimization is complicated by its large search space as well, which is non-linear in nature [26].

## 3 Kernel Descriptions and Qualitative Analysis

For our evaluation, we consider two versions of covariance computation based on Pearson’s method and Kendall’s method, as implemented in the open-source R statistics package [3]. We also create the third Kernel by modifying the second kernel based on Kendall’s method. Given two test data sets, represented by an  $n_X \times n$  matrix  $X$ , an  $n_Y \times n$  matrix  $Y$ , and pre-computed mean vectors  $\bar{x}$  and  $\bar{y}$  of length  $n_X$  and  $n_Y$ , respectively, the basic covariance computation (based on Pearson’s method) produces an  $n_X \times n_Y$  matrix  $C$  such that

$$C_{ij} \leftarrow \frac{1}{n-1} \sum_{k=1}^n (X_{ik} - \bar{x}_i) \cdot (Y_{jk} - \bar{y}_j)$$

In this section, we describe three kernels we considered, and explain their high-level characteristics.

### 3.1 Conventional Sequential Code

```
//p_x: a pointer for X
//p_y: a pointer for Y
//p_xm: a pointer for  $\bar{x}$ 
//p_ym: a pointer for  $\bar{y}$ 
//p_ans: a pointer for C
for(i = 0 ; i < n_X ; i++) {
    p_xx = &p_x[i * n];
    xxm = p_xm[i];
    for(j = 0 ; j < n_Y ; j++) {
        p_yy = &p_y[j * n];
        yym = p_ym[j];
        sum = 0.0;
        for(k = 0 ; k < n ; k++) {
            sum += (p_xx[k] - xxm) * (p_yy[k] - yym);
        }
        p_ans[i * n_Y + j] = sum / (n - 1);
    }
}
```

Code 1: C code for Kernel1

We show the C implementation of the basic covariance kernel based on Pearson’s method in Code 1. We refer to this code as “Kernel1.” “Kernel2” computes covariance using Kendall’s method and “Kernel3” is artificially created by modifying Kernel2. Kernel2 and Kernel3 are shown in Code 2. Kernel1 and Kernel2 codes are adopted from the R project source code [3].

In Kernel1, we can first subtract the mean vector from each matrix operand to remove the redundant subtracts. Then, transposing matrix Y converts this algorithm to a dense matrix multiplication problem, which is extensively studied and also there is a highly optimized BLAS library

```

//p_x: a pointer for X
//p_y: a pointer for Y
//p_ans: a pointer for C
for(i = 0 ; i < n_X ; i++) {
    p_xx = &p_x[i * n];
    for(j = 0 ; j < n_Y ; j++) {
        p_yy = &p_y[j * n];
        sum = 0.0;
        for(k = 0 ; k < n ; k++) {
            for(n1 = 0 ; n1 < n ; n1++) {
# if SIGN // Kernel2
                sum += sign((p_xx[k] - p_xx[n1])
                    * (p_yy[k] - p_yy[n1]));
# else // Kernel3
                sum += (p_xx[k] - p_xx[n1])
                    * (p_yy[k] - p_yy[n1]);
# endif
            }
        }
        p_ans[i * n_Y + j] = sum;
    }
}

```

Code 2: C code for Kernel2 and Kernel3

for the problem. Kernel2 and Kernel3 have more complex data access patterns but still can be optimized based on a cache blocking approach. Initially, *we intentionally ignore these particular optimization opportunities for the following two reasons*. First, we wish to stress the memory systems experimentally, and secondly, we wish to show the more typical and intuitive optimization process that will be common in practice. Then, if memory bandwidth turns out to be a performance bottleneck, we implement the blocking approach. Our focus is on highlighting the impact of architectural design trade-offs on performance and programmability. In particular, we do not intend to conclude which system is the “best” for computing covariance, nor do we claim to have implemented the best possible covariance code.

### 3.2 Basic Algorithmic Analysis

The memory footprint of all three kernels is  $O((n_X + n_Y) \times n)$  and the size of two input matrices are typically much larger than mean vectors or the output matrix. The computational complexity is  $O(n_X \times n_Y \times n)$  for Kernel1 and  $O(n_X \times n_Y \times n^2)$  for Kernel2 and Kernel3. While these kernels are compute-intensive in their asymptotic notations, if the entire memory footprint does not fit into the on-chip memory of the test systems, then these kernels can be bandwidth-bound. All three kernels have obvious  $n_X \times n_Y$  way parallelism as every pair of rows from matrix X and Y can be computed independently. Also, if we ignore the floating-point associativity issues, we can also trivially parallelize the innermost loop of Kernel1 and the second innermost loop of Kernel2 and Kernel3. For Kernel1, if we execute the code in a sequential way, there is higher tempo-

ral locality in the row data of matrix X than the row data of matrix Y. For Kernel2 and Kernel3, if we can place two rows from matrix X and Y on on-chip memory, we can perform  $O(n^2)$  computation over  $O(n)$  data without off-chip memory access. The total number of flops executed by Kernel1 is  $(n_X + n_Y) \times n + n_X \times n_Y \times n \times 2$ , and  $n_X \times n_Y \times n^2 \times 4$  for Kernel2 and Kernel3.

## 4 Baseline Architecture-specific Implementations

For subsequent evaluation, we create a basic parallel implementation for each architecture, described in this section. These implementations include “baseline” parallelization and tuning, meaning they include some degree of platform-specific tuning but are not extensively tuned. Again, as Section 3.1 states, our focus is on system evaluation and not on kernel optimization.

### 4.1 Intel Harpertown (2P) and AMD Barcelona (4P) Multicore Implementations

We can easily parallelize the outermost loop of all three kernels with OpenMP or pthreads for our 8 and 16 core systems, assuming a sufficiently large  $n_X$  ( $n_Y$ ). For Kernel1, we apply auto-vectorization with two directives, `#pragma unroll(16)` and `#pragma vector aligned`, achieving comparable performance to an intrinsics-based vectorization approach.

For Kernel2, the `sign()` function involves branches, lowering the performance significantly. We replace the branch with an SSE compare (e.g., `_mm_cmpgt_pd()` and `_mm_cmplt_pd()`) and bitwise operations (e.g., `_mm_and_pd()` and `_mm_or_pd()`). The Intel icc compiler fails to perform this replacement automatically, and so we hand-code this translation to use SIMD intrinsics. The Kernel3 code can be trivially vectorized in the same way.

For the 4P Barcelona system, which is NUMA-based, we replicate the input matrices to all four chip’s local DRAM, and pin the threads to each core. The replication cost can be amortized with the multiple reads, and this optimization maximizes the available bandwidth while minimizing the interference.

Even though our test kernels are asymptotically compute-intensive, if the input matrices do not fit into the on-chip cache memory, these algorithms can be bandwidth-bound. A blocking approach can reduce the amount of off-chip data transfer at the cost of increased implementation complexity. Also, for the Harpertown and Barcelona processors, selecting the optimal block size requires exhaustive search over parameter space as it is a complex function of

the multiple levels of cache hierarchy and their size and associativity. This exhaustive search is beyond the scope of our work and we find the block size based on heuristics.

## 4.2 STI Cell/B.E. (2P) Implementation

The Cell/B.E. implementation resembles the 4P AMD Barcelona implementation, though it provides an additional opportunity for fine-tuning owing to the higher level of control over on-chip memory supported by the architecture. In particular, observe that a row data of matrix  $X$  has, assuming the given loop order, a higher temporal locality than a row data of matrix  $Y$ . Thus, we can assign a larger buffer for matrix  $X$  than  $Y$ . Furthermore, to reduce the bandwidth requirement even when a single row does not fit into the local store, we allocate additional small buffers for streaming. In this case, we read data from the larger buffer for matrix  $X$  and  $Y$  for accessing the initial part of the row (which fits into the local store), and then we switch to the streaming mode with the smaller buffers for the remaining.

However, fine-grained control over on-chip memory significantly increases the coding complexity, especially when the on-chip memory requirement varies as a function of an input data size. A blocking approach, even though it adds additional complexity in high level, fixes the on-chip memory requirement regardless of an input data size. Accordingly, a blocking approach can reduce the coding complexity for the Cell/B.E. in addition to the improved performance. For the Cell/B.E., the impact of different block size is easier to understand owing to its simple memory subsystem. Larger block height reduces the amount of traffic whereas larger block width increases the iteration count of the innermost loop to improve the compute efficiency. We can also simply pick the largest block size that fits into the local store instead of considering different cache sizes in the memory hierarchy.

## 4.3 NVIDIA Tesla C1060 Implementation

For the NVIDIA Tesla C1060,  $n_X \times n_Y$ -way parallelism may not be sufficient for practical data set sizes. Even when  $n_X \times n_Y$  is very large, having every thread processes a distinct pair of rows can lead to poor bandwidth utilization (no coalescing in data transfer) or low on-chip cache utilization (no data sharing). For Kernel1, we partition the innermost loop with chunks of size 16 elements (a half warp, as high memory bandwidth utilization is achieved when the memory accesses from a half warp can be coalesced [20]). Each thread in a half warp processes one element out of 16 elements in a chunk to maximize the coalescing. For Kernel2 and Kernel3, we partition the second innermost loop identical to the case of Kernel1. In this case, every thread in a half warp traverses the same row data in a synchronized

way (in the innermost loop of Code 2, array index  $k$  remains constant and only array index  $n1$  changes. In our optimized code, every thread in a same half warp accesses  $p_{xx}$  and  $p_{yy}$  with same  $n1$  but different  $k$ ), and we can use the on-chip shared memory to exploit this fact. As each thread accesses the same data element, we can use the shared memory’s broadcasting mechanism as well.

One critical issue is the  $sign()$  function, which involves branch instructions. The NVIDIA CUDA compiler replaces branch instructions with predicates when the number of instructions controlled by the branch is equal to or less than the threshold value (4 or 7 instructions) [20]. Therefore, by using the CUDA framework, we do not need to manually optimize for  $sign()$  function, as we did on the Intel Harpertown, AMD Barcelona, and Cell/B.E. platforms. Optimization for the Tesla C1060 is more involved in high level, but simpler to program than the Cell/B.E. for these kernels.

Also, the proper use of on-chip cache memory significantly reduces the bandwidth requirement, and Kernel2 and Kernel3 become compute-bound even without explicit blocking.

## 4.4 Quantitative Comparison of Implementation Costs

	kernel code size (# of lines)	approximate coding time
Harpertown and Barcelona - initial	335	1 day
Harpertown and Barcelona - blocking	419	2 days
Cell/B.E. - initial	1620	7 days
Cell/B.E. - blocking	1004	2 days
Tesla C1060 - initial	52 (Kernel1) + 97 (Kernel2/3)	2 days
Tesla C1060 - blocking	88 (Kernel1)	1 day

Table 2: Quantitative comparison of implementation costs in terms of code size and implementation time. This excludes the code for the kernel invocation and the residual part computation.

Table 2 summarizes the comparison. For the Cell/B.E., a blocking approach fixes the local store space requirement regardless of an input data size, and simplifies the coding in addition to the improved performance. We can also identify that the code size for the Tesla C1060 is significantly smaller than the other architectures. For the Tesla C1060, the challenge is in extracting additional parallelism and best exploiting the memory subsystem (based on data access coalescing and broadcasting mechanism and efficient use of the shared memory).

## 5 Experimental Evaluation: Measurements and Analysis

Recall the evaluation platforms from Table 1. To measure the sustained bandwidth of the 2P Harpertown system, we use PAPI [2] and count the number of memory bus transactions (event code 0x40001045 in PAPI). For the 4P Barcelona system, we use AMD CodeAnalyst [1] and count the number of DRAM accesses (event select code 0xE0 for the AMD Barcelona’s event select register). For the systems with the Cell/B.E. processors, we attach counter variables to every DMA memory requests and ignore the PPE initiated traffic. For the Tesla C1060, we estimate the total bandwidth requirement using the following equations:  $n_X \times n_Y \times n \times \text{sizeof(float or double)} \times 2$  for Kernel1 and  $n_X \times n_Y \times n \times n \times \text{sizeof(float or double)} \times 2 \times \frac{1}{16}$  (a half warp width, owing to data sharing) for Kernel2 and Kernel3. For the NUMA-based AMD Barcelona and Cell/B.E. architectures, we replicate matrix X and Y, which are read multiple times, for higher bandwidth utilization. This replication cost is included for the timing, and the off-loading overhead to the device memory in the Tesla C1060 is also included.

### 5.1 Kernel1

The top half of Figure 2 depicts the sustained Gflop/s and bandwidth for Kernel1 in single-precision with the initial implementation. Although the algorithm is computationally intensive, the performance is bounded by memory bandwidth since the entire data do not fit into the on-chip memory. The Tesla C1060 benefits from its high bandwidth to on-board DRAM, but the sustained bandwidth is lower than the theoretical peak and varies significantly for different input matrix sizes. The off-loading overhead accounts for 18% (for the smallest matrix) to 2.4% (for the largest matrix) of the total execution time. The QS20 and QS22 blades achieve the highest bandwidth utilization on average across the different values of  $n$  owing to their DMA based data transfer mechanism with a large chunk size. The QS20 blade (with Rambus XDR) achieves higher bandwidth utilization than the QS22 blade (with DDR2). The 4P Barcelona system achieves significantly higher sustained bandwidth and bandwidth utilization than the 2P Harpertown system. This exemplifies the scalability benefit of NUMA architecture. The AMD Barcelona has optimized the memory access scheduling algorithms for interleaved streaming accesses, and this also contributes to higher bandwidth utilization. However, the 2P Harpertown system delivers higher flop rates per unit bandwidth consumption owing to the large shared cache memory.

The blocked implementations<sup>3</sup> yield significantly better results than the initial implementation, as shown in the bot-

<sup>3</sup>We find the demand paging related performance issue with the

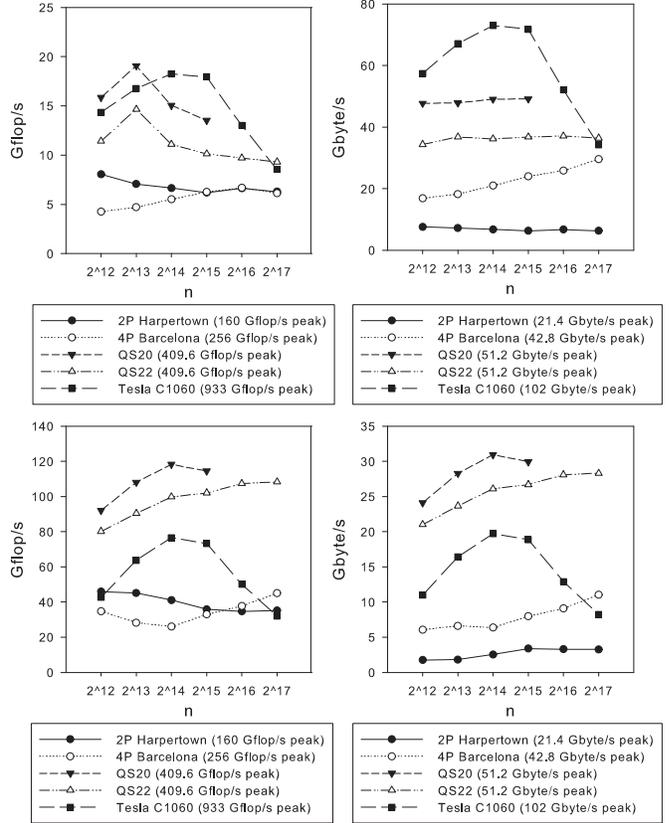


Figure 2: Sustained Gflop/s (left) and bandwidth utilization (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel1 (single-precision). Missing points for the QS20 are due to memory allocation failure. Here,  $n_X = n_Y = 1024$ .

tom half of Figure 2, but still deliver significantly lower performance than the theoretical peak. In the case of the x86 architectures, there are a number of possible explanations. First, we need to tune the blocking with respect to the different sizes and associativities at all levels of the cache hierarchy to achieve higher performance. This task would be daunting task even for skilled programmers. Secondly, the blocked implementation may interact, and may even interfere, with the various hardware mechanisms in an unintentionally negative way. For example, the blocked version has a more complex memory access pattern, which may reduce the effectiveness of the hardware prefetchers. Thirdly, the behavior of the memory system mechanisms are com-

NUMA-based implementations later and update the performance results of the AMD Barcelona and the QS20 accordingly (60%-80% and 5%-20% speedup for the Barcelona and the QS20 in the case of Kernel1 with blocking, respectively. The speedup is insignificant in other cases). The QS22 blade is not accessible at this time, and for the QS22, we expect the speedup comparable to the QS20.

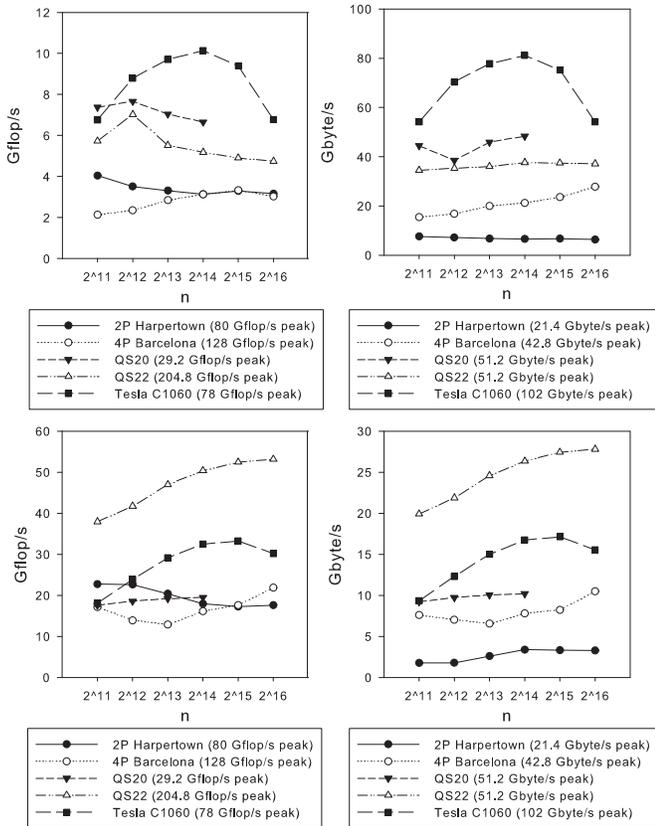


Figure 3: Sustained Gflop/s (left) and bandwidth (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel1 (double-precision). Missing points for the QS20 are due to memory allocation failure. Here,  $n_X = n_Y = 1024$ .

plex and challenging to reason about. For instance, on the Harpertown, data in DRAM is first read into the lower level (L2) cache, whereas it is read into the highest level (L1) cache first and moved to the non-inclusive L2 and L3 caches (when the cache line is evicted) on the Barcelona. All these differences can affect the performance in non-intuitive ways and this imposes challenges to a programmer if one wishes to extract the highest achievable flop rates out of the chip.

In contrast, for the Cell/B.E. based systems, it is significantly easier to understand the data transfer related performance issues owing to its simple architecture. Still, to achieve the highest flop rate, programmers need to consider their code at the assembly level. Each iteration of the innermost loop in Kernel1 requires two vector loads, one vector stores, two address increments, and one vector FMA (fused-multiply-and-add) instructions. As a result, the fixed-point instructions and load/store instructions can become a performance bottleneck. Extensive low level tuning is required

to balance the even and odd pipeline and minimize the address calculation overhead.

For the Tesla C1060, the delivered performance is lower than 10% of its theoretical peak even in the best case. The off-loading overhead (11%-70% of the total execution time, which increases as  $n$  decreases), integer instructions for address increments, and the lack of additional multiply instructions to feed the SFU lower the deliverable performance. Also, to load data to the block array in the shared memory, the Tesla C1060 needs to calculate an address and issue a load instruction for every single real number even though DRAM access latency can be in principle efficiently hidden with the hardware-multithreading mechanism; thus the device memory to the shared memory traffic cannot be perfectly overlapped with the computation as is the case of the Cell/B.E. The Tesla C1060 architecture is less transparent than the Cell/B.E.'s, and so the performance impact of the tunable parameters (e.g. thread block size, block width and height in a blocking approach, and loop unrolling factors) are more difficult to predict, thereby requiring explicit search and tuning.

Figure 3 shows the results for double-precision. Interestingly, we can see that the sustained bandwidth for the Tesla C1060 is higher than the single-precision case, largely due to the increased granularity of data transfer from 64 byte (16 threads in a half warp  $\times$  4 byte floating point number) to 128 byte (16  $\times$  8 byte floating point number). In the case of double-precision, the QS22 blade delivers higher performance than the QS20 blade, and the QS20 blade becomes compute-bound. We can also note that QS20 and the Tesla C1060 achieve a significantly higher fraction of its theoretical peak than the single-precision case. This shows that the QS20 and the Tesla C1060's peak double-precision performance is easier to achieve than its single-precision counterpart.

## 5.2 Kernel2

Figures 4 and 5 summarize the results for Kernel2 in single- and double-precision, respectively. This kernel is highly compute-intensive, and if a pair of two rows can fit into the on-chip memory, Kernel2 will be compute-bound even without blocking. For single-precision, the Tesla C1060 delivers nearly 100 Gflop/s. Still, this number is significantly lower than the advertised peak numbers, for similar reasons discussed in the case of Kernel1 in addition to *sign()* function. Kernel2's invocation of *sign()* is replaced with predicated instructions; however, computing predication also requires additional cycles. In addition, as the SP has only one execution pipeline and does not have separate integer execution units, performance is more susceptible to the mix of integer instructions than on other architectures. For the QS20 and QS22 blades, the kernel is compute-

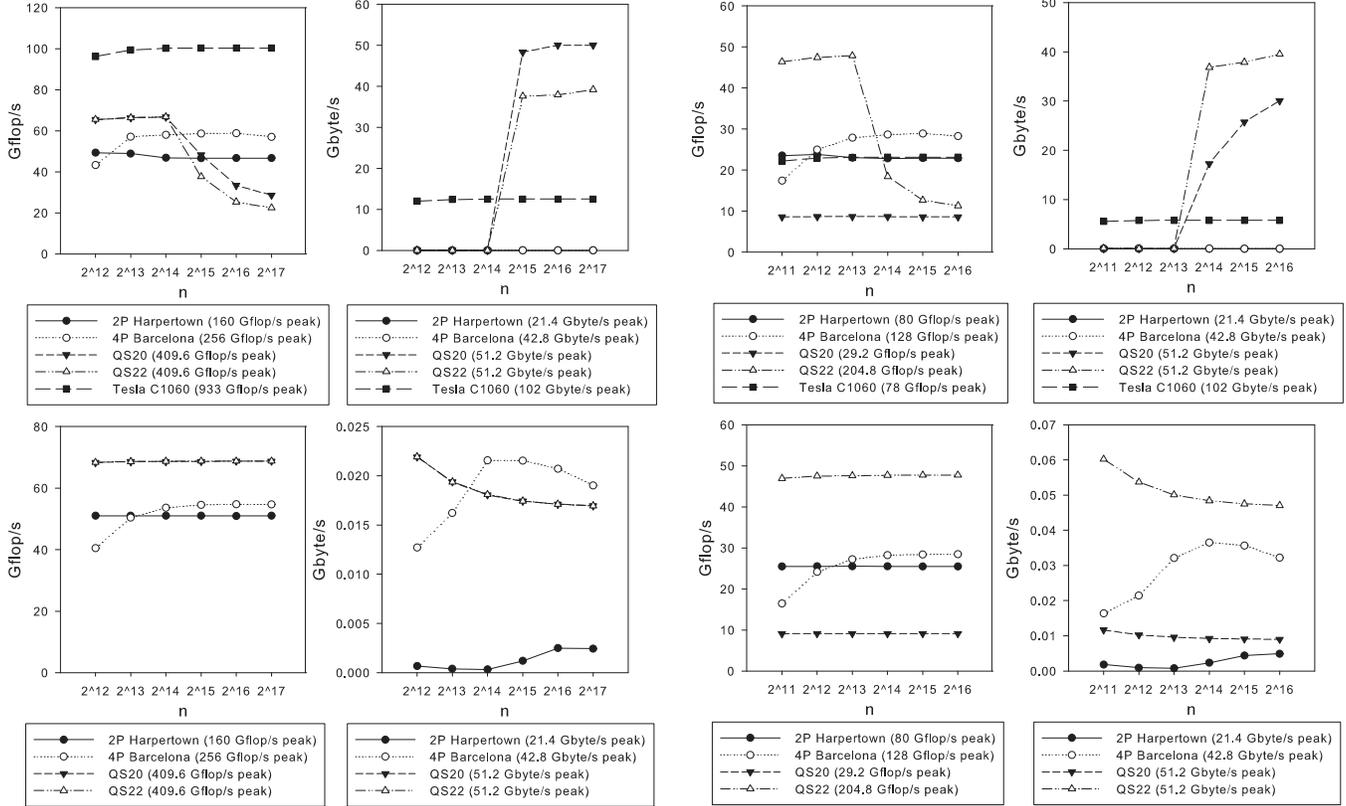


Figure 4: Sustained Gflop/s (left) and bandwidth (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel2 (single-precision). Here,  $n_X = n_Y = 128$ .

Figure 5: Sustained Gflop/s (left) and bandwidth (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel2 (double-precision). Here,  $n_X = n_Y = 128$ .

bound for small  $n$  and bandwidth-bound for large  $n$  without blocking. In contrast, the QS20 and QS22 blades become compute-bound even with large  $n$  with blocking. In the bandwidth-limited cases, the QS20 blade achieves nearly full bandwidth utilization while the QS22 blade delivers approximately 80% of the theoretical peak bandwidth. The QS20 blade uses XDR DRAM and the QS22 blade uses DDR2 DRAM, which explains the reason for the different sustained bandwidth.

For the Tesla C1060, the kernel becomes compute-bound, even without blocking, because of the efficient use of the shared memory. Indeed, blocking does not improve performance, and so we need not consider it further.

### 5.3 Kernel3

Kernel3 is highly floating-point intensive, and its performance is given in Figure 6 and 7. For single-precision, the QS20 blade, the QS22 blade, and the Tesla C1060 card deliver comparable performance for small  $n$ . The QS20 and QS22 blade's performance drops sharply with large  $n$

without blocking, but the flop rates remains nearly constant with blocking. Considering that our kernel has more adds and subtracts than multiplies, the QS20 and QS22 blades demonstrate near the maximum achievable floating-point performance. Yet, the performance for the Tesla C1060 is only one-quarter of its advertised peak performance. For double-precision, the Tesla C1060 delivers over half of its peak flop rates in comparison to one-quarter in the single-precision case as double-precision SP can be solely used for floating-point operations and single-precision SPs can be exploited for integer instructions (e.g. address calculation).

## 6 Conclusions

Based on our detailed comparison of multicore processors and accelerators, we can draw several general conclusions. First, a hardware-multithreading approach lowers programming complexity by freeing programmers from latency issues at the expense of increased requirements for

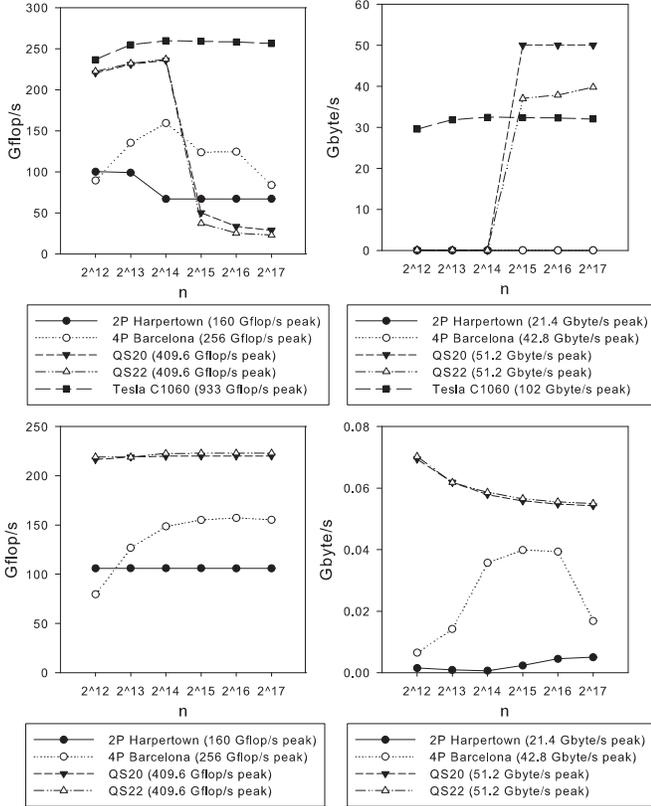


Figure 6: Sustained Gflop/s (left) and bandwidth (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel3 (single-precision). Here,  $n_X = n_Y = 128$ .

parallelism. Second, NUMA has benefits over UMA for data-intensive algorithms if the underlying system or programmer can properly handle data placement and thread binding. Third, an explicit DMA-based data transfer mechanism can effectively achieve high bandwidth utilization, at the cost of higher coding complexity. Fourth, in the case of the Cell/B.E., blocking can also reduce the programming complexity as the programmer is forced to optimize the program with the fixed size on-chip memory. Fifth, increasing raw floating-point performance only may not lead to overall system performance enhancement in many cases, and the off-chip memory bandwidth is critical even when the algorithm appears to be compute-bound in asymptotic notation. Integer execution units are also important to achieve the peak flop rates as well. Sixth, the Cell/B.E.'s simple memory subsystem increases the coding complexity to implement the first correct version of the code with proper data transfer mechanisms. Yet, the subsequent optimization process is easier and more intuitive. The opposite holds for the x86 based architectures. This paper provides quantitative support for these conclusions, presented in a tutorial-

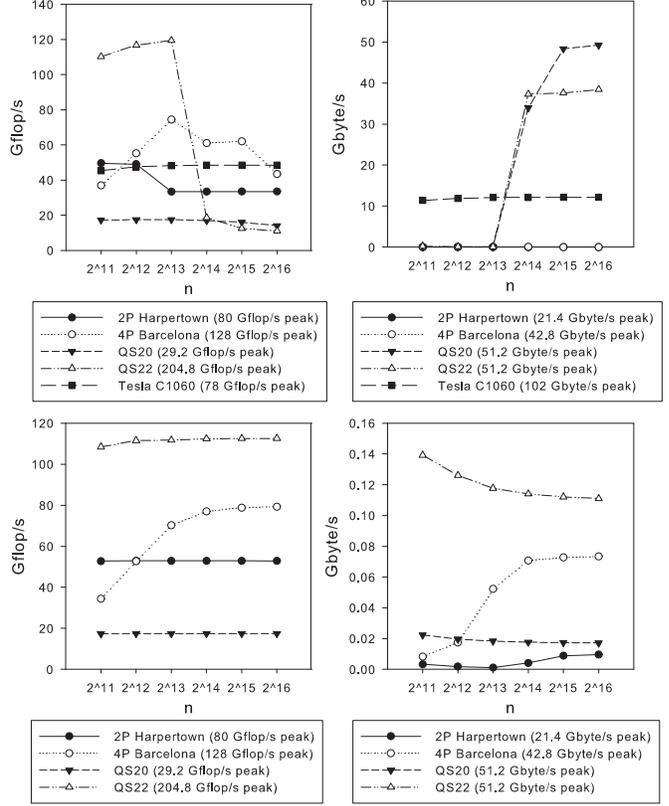


Figure 7: Sustained Gflop/s (left) and bandwidth (GB/s) (right) for the initial (top) and the blocking based (bottom) implementations of Kernel3 (double-precision). Here,  $n_X = n_Y = 128$ .

style discussion that we hope can be used to guide future tuning and architecture selection efforts. Looking forward, we intend to evaluate different hardware systems with more complex and irregular kernels, as well as real-world applications.

## Acknowledgments

This work was supported in part by an IBM Shared University Research (SUR) award and NSF Grants CNS-0614915 and DBI-0420513. We acknowledge NVIDIA Corp. for its donation of GPU cards and the professor partnership award, and the Sony-Toshiba-IBM Center of Competence for use of Cell/B.E. resources that have contributed to this research.

## References

- [1] AMD CodeAnalyst, 2009. <http://developer.amd.com/cpu/CodeAnalyst>.

- [2] PAPI, 2009. <http://icl.cs.utk.edu/papi>.
- [3] The R project for statistical computing, 2009. <http://www.r-project.org/>.
- [4] AMD Corporation. *Software Optimization Guide for AMD Family 10h Processors*, 3.06 edition, Apr. 2008.
- [5] O. Bockenbach, M. Knaup, and M. Kachelriess. Implementation of a cone-beam backprojection algorithm on the Cell Broadband Engine processor. In *Proc. SPIE Medical Imaging*, San Diego, CA, Feb. 2007.
- [6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.
- [7] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. Accelerating compute-intensive applications with GPUs and FPGAs. In *Proc. 6th IEEE Symp. on Application Specific Processors (SASP)*, Anaheim, CA, Jun. 2008.
- [8] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation—a performance view. *IBM Journal of Research and Developments*, 51(5):559–572, Sep. 2007.
- [9] P. Gepner, D. L. Fraser, and M. F. Kowalik. Second generation quad-core Intel Xeon processors bring 45 nm technology and a new level of performance to HPC applications. *Lecture Notes in Computer Science*, 5101:417–426, 2008.
- [10] Intel Corporation. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Nov. 2007.
- [11] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. *ACM SIGARCH Computer Architecture News*, 36(3):39–50, Jun. 2008.
- [12] D. Jimenez-Gonzalez, X. Martorell, and A. Ramirez. Performance analysis of Cell Broadband Engine for high memory bandwidth applications. In *Proc. 7th IEEE Int’l Symp. on Performance Analysis of Systems and Software (ISPASS)*, San Jose, CA, Apr. 2007.
- [13] M. G. Kendall. A new measure of rank correlation. *Biometrika Trust*, 30(1):81–93, Jun. 1938.
- [14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar. 2008.
- [15] S. A. McKee, W. A. Wulf, J. H. Aylor, R. H. Klenke, M. H. Salinas, S. I. Hong, and D. A. B. Weikle. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, Nov. 2000.
- [16] J. S. Meredith, S. R. Alam, and J. S. Vetter. Analysis of a computational biology simulation technique on emerging processing architectures. In *Proc. 6th IEEE Int’l Workshop on High Performance Computational Biology (HICOMB)*, Long Beach, CA, 2007.
- [17] O. Mutlu and T. Moscibroda. Enhancing the performance and fairness of shared DRAM systems with parallelism-aware batch scheduling. In *Proc. 35th Ann. Int’l Symp. on Computer Architecture (ISCA)*, Beijing, China, Jun. 2008.
- [18] C. Natarajan, B. Christenson, and F. Briggs. A study of performance impact of memory controller features in multi-processor server environment. In *Proc. 3rd Workshop on Memory Performance Issues (WMPI)*, Munich, Germany, Jun. 2004.
- [19] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *ACM Queue*, 6(2):40–53, Mar. 2008.
- [20] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*, 2.0 edition, Jun. 2008.
- [21] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Kruger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, Mar. 2007.
- [22] L. A. Polka, H. Kalyanam, G. Hu, and S. Krishnamoorthy. Package technology to address the memory bandwidth challenge for tera-scale computing. *Intel Technology Journal*, 11(3), 2007.
- [23] N. Rafique, W. T. Lim, and M. Thottethodi. Effective management of DRAM bandwidth in multicore processors. In *Proc. 16th Int’l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Brasov, Romania, Sep. 2007.
- [24] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proc. 27th Ann. Int’l Symp. on Computer Architecture (ISCA)*, Vancouver, Canada, Jun. 2000.
- [25] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Salt Lake City, UT, Feb. 2008.
- [26] S. Ryoo, C. I. Rodrigues, S. S. Stone, J. A. Stratton, S. Ueng, S. S. Baghsorkhi, and W. W. Hwu. Program optimization carving for GPU computing. *Journal of Parallel and Distributed Computing*, 68(10):1389–1401, 2008.
- [27] T. Saidani, S. Piskorski, L. Lacassagne, and S. Bouaziz. Parallelization schemes for memory optimization on the Cell processor: a case study of image processing algorithm. In *Proc. Workshop on memory performance (MEDEA)*, Brasov, Romania, Sep. 2007.
- [28] O. Schenk, M. Christen, and H. Burkhart. Algorithmic performance studies on graphics processing units. *Journal of Parallel and Distributed Computing*, 68(10):1360–1369, 2008.
- [29] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerma, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3), Aug. 2008.
- [30] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proc. Int’l Conf. on High Performance Computing and Networking (SC)*, Austin, TX, 2008.
- [31] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. Int’l Conf. on High Performance Computing and Networking (SC)*, 2007.