RC25070 (W1010-125) October 27, 2010
Computer Science

IBM Research Report

An Experimental Analysis of General Purpose Computing with Commodity Data-Parallel Multicore Processors

Rebecca Collins, Cheng-Hong Li, Luca P. Carloni
Department of Computer Science
Columbia University
New York, NY 10027 USA

Kevin J. Nowka
IBM Research Division
Austin Research Laboratory
11501 Burnet Road
Austin, TX 78758 USA

Eugen Schenfeld
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA

An Experimental Analysis of General Purpose Computing with Commodity Data-Parallel Multicore Processors

Rebecca Collins
Columbia University

Cheng-Hong Li
Columbia University

Luca P. Carloni
Columbia University

Kevin J. Nowka
IBM Corporation

Eugen Schenfeld
IBM Corporation

ABSTRACT

Multicore processors have recently emerged as a better solution to leverage the benefits of semiconductor technology scaling than traditional architectures based on a single CPU. Meanwhile, graphics processing units (GPUs) are increasingly used not only for graphics rendering but also for general purpose computing. High-end GPUs also follow the trend of integrating multiple processing units on a single chip while adding high memory-access bandwidth. Exploiting the computational potential of multicore architectures requires new programming paradigms and tools that are optimized for parallel computing rather than conventional (single-processor) sequential programming. We select three non-graphics benchmark applications to conduct head-to-head performance comparisons between two data-parallel multicore processors: a state-of-the-art GPU, the NVIDIA GeForce 8800 GTX, and a leading heterogeneous multicore processor, the IBM Cell Broadband Engine. We also present a preliminary analysis of the trade-offs between programmability and code efficiency for these benchmarks by comparing implementations that are based on processor-specific software development environments with those obtained with a portable multi-core development platform.

1. INTRODUCTION

Thanks to continuous progress in semiconductor technology a single chip will soon host hundreds of processing cores that may either share the same architecture (homogeneous cores) or be specialized for particular tasks (heterogeneous cores) [11]. As systems based on multicore processors are soon to deliver tens of tera operations per seconds, the challenge from the programmers' perspective is to harness this unprecedented computational power efficiently. This will require exploiting parallelism at various levels of granularity: not only instruction-level parallelism, the main target of traditional microprocessors, but also thread-level, coarse-grained task-level, and data parallelism [16].

High-end Graphic Processing Units (GPUs) are arguably the first example of multi-core chips having hundreds of processing cores. Originally designed for the specific graphics-rendering purpose, GPUs have recently been considered for general-purpose computing [2] as well as high-performance computing [7]. The computational capabilities of a GPU, measured by the traditional metrics of graphics performance, have compounded at an average yearly rate of 1.7$\times$ (pixels/second) to 2.3$\times$ (vertices/second), a growth rate outpacing Moore's Law as applied to traditional microprocessors [30]. The reason for this disparity is that traditional CPU's hardware has been optimized for a sequential programming model: a large fraction of their transistors and wires implement complex control functionality such as branch prediction to extract instruction-level parallelism and cache memories to minimize the latency of memory access. In contrast, the bulk of the transistors in a GPU chip are used to exploit the high levels of data parallelism in graphics applications and to maximize the bandwidth of many simultaneous memory accesses [29]. However, there are many important non-graphics applications that map well to the GPU's dense array of stream processors or that require sifting through large quantities of data, thus mapping well to the GPU's high-bandwidth memory subsystem. In fact, it is reported that for certain applications GPUs can be as much as 20 to 100 times faster compared to state-of-the-art CPUs running optimized code [22]. Furthermore, this computational power is both available and inexpensive as GPU chips are used in pervasive off-the-shelf graphics cards for personal computer and video game consoles.

As discussed in [30], the next-generation of GPU architects face the challenge of improving the programmability and generality of GPU architectures, but without sacrificing the specialized performance that have made them successful. Meanwhile, the competition is growing as other commodity data-parallel multicore processors are emerging. Among these, the Cell Broadband Engine (BE) [19, 20, 31], which was originally designed for the PlayStation 3 video game console, is now used for both general-purpose enterprise applications [13] and high-performance computing applications [3]. The Cell BE is a heterogeneous multicore processor that combines GPU-like features (such as an array of SIMD processors and high-bandwidth off-chip communication links) with a traditional CPU processor core and on-chip cache. These two architectures have different strengths and weaknesses and are each better suited to different types

Rebecca Collins, Cheng-Hong Li, and Luca P. Carloni are with the Department of Computer Science at Columbia University, New York, NY 10027, {rlc2119,cheli,luca}@cs.columbia.edu. Kevin J. Nowka is with IBM Austin Reseach Laboratory, Austin, TX 78758 nowka@us.ibm.com. Eugen Schenfeld is with IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, eugen@us.ibm.com.

	IBM Cell BE [31]	NVIDIA 8800 GTX [9]
processing core #	2× (1 PPE + 8 SPEs)	128 SPs
technology (nm)	90	90
core clock rate (GHz)	3.2	1.35
transistors (M)	2× 239	690
main memory (MB)	1000	768
mem. bandwidth (GB/s)	25.6	86.4

Table 1: Main features of the two hardware execution platforms.

of applications. It is an open question which data-parallel multicore architecture can be considered a "better" architecture for general-purpose programming as a whole.

Contributions. This paper presents initial experiments of using novel programming tools with two leading data-parallel multicore processors available today. Our approach is to explore a few benchmarks which range from computation intensive (Monte Carlo simulation for option pricing) to communication intensive (sorting). We consider the use of novel programming tools, which promise to limit the complexity and offer the simplicity and portability to work with emerging multicore processing chips. Our goal is not to achieve the fastest possible benchmark implementation, but instead to understand how much performance is lost when we move from processor-specific software development kits to high-level portable multi-core development platform.

Our second goal is to compare the performance of Cell and the NVIDIA GPU. The comparison is not yet conclusive as more detailed examination is needed. So far, we conclude that for communication-rich applications the Cell BE is better, while the NVIDIA GPU is stronger for computation-rich applications.

These comparisons (high-level vs. low-level programming tools, and Cell vs GPU architectures) are complementary since the programming tools must expose and utilize the features of the platforms. Likewise, the architectures must provide memory, communication, and computing support for general-purpose applications. As both programming tools and multicore chips evolve, we expect that each will influence the other.

2. HARDWARE EXECUTION PLATFORMS

The two computation platforms used in our experiments are the IBM Cell blade QS20 equipped with two Cell BE multicore processors, and a GPU graphics card with the NVIDIA GeForce 8800 GTX. Table 1 presents the main hardware features of these platforms. We briefly describe each of them next.

2.1 Cell Broadband Engine

The Cell Broadband Engine (BE) [19, 20, 31] was jointly designed by IBM, Sony and Toshiba for the PlayStation 3 game console (PS3). Although Cell is well known for its use on the PS3, it is also used in servers designed for high-performance computing applications such as IBM's QS20 and QS21 Cell blades [28] and Mercury System's Dual Cell-Based servers [5].

The Cell processor is a heterogeneous multicore processor featuring eight synergistic processing units (SPE) cores and one dual-threaded 64-bit PowerPC Element (PPE), which is used as the supervising core. Each SPE contains a *single instruction multiple data* (SIMD) processor operating on entries of 128-bit registers that, for instance, can each be organized as a vector of four 32-bit integers. Each SPE core operates on a *local store (LS)* memory of $256KB$ that is used for both data and code. A local store is not a traditional cache memory as it is software managed. Data is transferred between cores via direct memory access (DMA).

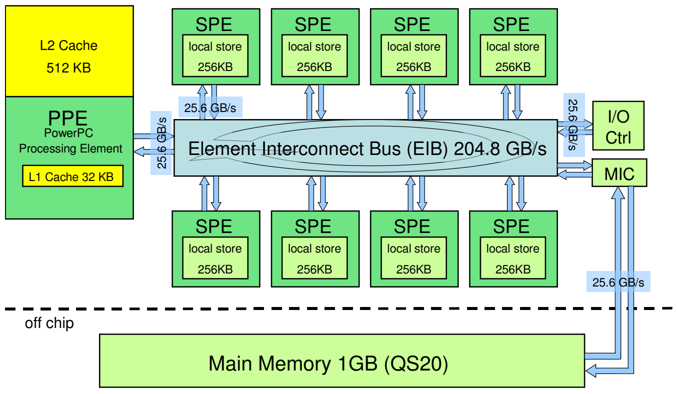

Figure 1: Cell BE Architecture.

The Cell BE has a powerful on-chip network for inter-core communication, called the Element Interconnect Bus (EIB), made up of four circuit-switched rings [10, 21]. Two rings transfer data in one direction and two transfer data in the opposite direction. Only data transfer circuits that are less than half of a ring away are scheduled. Each ring supports up to three concurrent, non-overlapping data transfers. The EIB supports an on-chip communication bandwidth of over 200 GB/s. The main memory uses an XDR RAM interface with a 25.6 GB/s bandwidth.

Our Cell BE experiments were performed on an IBM QS20 Cell Blade with a 1GB of memory and two Cell BE chips directly connected by a 20 GB/s cache coherent bidirectional link [28].

2.2 NVIDIA GeForce 8800 GTX

The second hardware platform used in our experiments is the NVIDIA GeForce 8800 GTX GPU. This is a high-end programmable GPU with massive floating-point computation performance. It is this programmability which makes it also suitable for general-purpose computing (hence, the acronym GPGPU) [22].

Figure 2 shows the high-level architecture of the GeForce 8800 GPU. It has 128 programmable processing units, called stream processors (SP), running at a clock rate of 1.5GHz. The 768MB external memory is connected to the SPs by several links with an aggregated maximum bandwidth of 86.4GB/sec. The SPs are divided into 16 groups, called multiprocessors, each with 8 SPs. The SPs in one group execute instructions in a SIMD fashion, i.e. at every clock cycle they execute the same instruction on different data. If a branch instruction changes the fetch direction of some, but not all of the SPs of the same group, the execution of instructions at the two different basic blocks will be serialized (leading to some SPs to stall). The SPs of the same multiprocessor

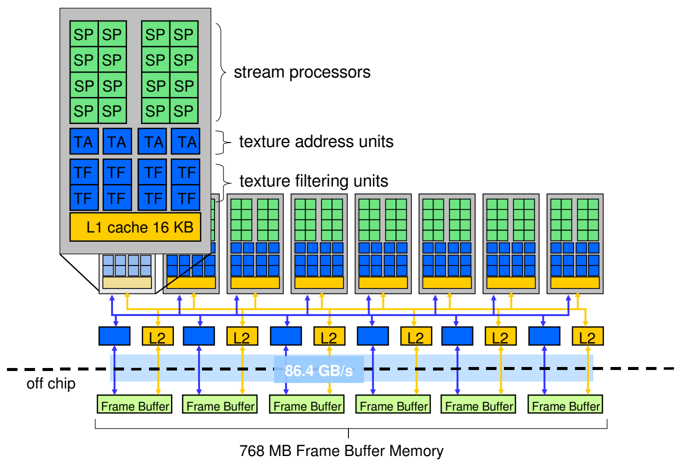

Figure 2: NVIDIA GTX 8800 Architecture

communicate with each other through on-chip shared memory. Global communication between processing units across multiprocessor boundaries is only possible through a shared location in the the external memory. This point is critical for the performance of applications with communication patterns requiring complex communications and is an important difference between this GPU and the Cell BE. Such difference, if exploited by the programming tools, leads to performance difference for certain applications.

For simplicity, in the following discussions we refer NVIDIA GeForce 8800 GTX as G80.

3. PLATFORMS FOR SOFTWARE DEVELOPMENT

Multiple software platforms can be used to program the Cell blade and G80. Each architecture has a native software development toolset (the Cell SDK and the NVIDIA CUDA, respectively). These give good control of the underlying hardware resources at a cost of requiring the programmer to take care of explicit thread and memory management. Software tools such as RapidMind abstract the underlying hardware details in exchange of providing less control on the underlying hardware platform resources

3.1 Cell BE SDK

The Cell BE Software Development Kit (Cell SDK) is collection of tools and libraries for programming the Cell. We use Version 2.1 of the Cell SDK for all of our experiments [4, 18]. The SDK includes a C/C++ compiler and programs can be written in standard C/C++. The code for the PPE is written separately from the code for the SPEs and during compilation different libraries are linked to each code set.

Thread Management. The main execution of an application as well as all I/O operations are handled by the PPE. From the PPE, threads are launched on the SPEs, which handle the core of the computation. The SPE consists of three tightly coupled units: the synergistic processing unit (SPU), the local store (LS), and the memory flow controller (MFC). The SPU is a SIMD processor with an instruction set architecture optimized for compute-intensive and media applications: It operates only on instructions and data in the associated local store. Decoupling the SPU from other aspects of the system provides a very deterministic processing environment for the programmer [19].

DMA. The bulk of data movement between the local store of an SPE and the main memory or the local store of another SPE takes place via asynchronous, coherent direct memory access (DMA) transfers under the supervision of a DMA controller in the MFC unit. Since local stores are not cache memories, data consistency must be explicitly managed. Data movement and synchronization are initiated by using MFC commands. Either the local SPU or another processor in the system (the PPE or another SPU) can issue such MFC commands. The PPE may also send and receive small messages from the SPEs with mailbox transfers. The SPEs cannot send each other mailbox messages, but there is a mechanism for exchanging signals.

Vector Operations. The SPU provides the programmer with 128 128-bit SIMD registers. The large number of registers facilitates efficient instruction scheduling and also enables important optimization techniques such as loop unrolling [4]. The Cell SDK provides a library of vector operations that expose the SIMD capabilities of the SPEs. The SPEs can perform both mathematical vector operations and conditional operations, for example, using a mask to select between the elements of two vectors.

3.2 NVIDIA CUDA

The *Compute Unified Device Architecture (CUDA)* is a programming interface and environment developed by NVIDIA for general-purpose programming of its own GPUs [6]. Together with its run-time library and tools, CUDA allows programmers to use the C programming language with simple extensions to write general-purpose software on GPU. Additionally, the CUDA library provides basic functions to allow programmers to access the specialized hardware on the GPU (for example, the texture memory). At a higher level it also offers optimized library for scientific computing, like the CUDA FFT library that we used in some of our experiments.

A CUDA program is launched on the host CPU and it distributes the GPU execution code and the data among the GPU devices over the system bus. The computation tasks to be executed on the GPU are implemented in C functions with special labels (part of the extended C language). Each invocation of such functions triggers CUDA's run-time backend to create threads running on the GPU processing units. Programmers have direct control over the number of threads to be created and their division within and across the multiprocessor groups. The remaining part of a CUDA program, which runs on the host CPU, is responsible for transferring data between GPU's on-board memory and the main memory.

3.3 RapidMind

RapidMind is a high-level data-parallel tool for programming multicore processors, including the NVIDIA and ATI GPUs as well as the Cell BE [25]. RapidMind adopts a *single program multiple data* (SPMD) streaming programming model where a single program can operate concurrently on an array of data [27].

RapidMind grew out of Sh, a tool for programming GPUs intended to both unify shader programs with their host programs and provide a more general-purpose programming platform for GPUs than was previously available [24].

RapidMind provides C++ libraries that add a few new types: `Array`, `Value`, and `Program`. A RapidMind program is called with a RapidMind array as input, and the program executes separately on each array element. For example, consider the following snippet of C++ code:

```
main {
   int i, a[N],b[N];
   // ... initialize a[] and b[] ...
   for(i=0; i<N; i++) {
      a[i] *= b[i];
   }
}
```

The code above is rewritten in RapidMind by first creating a RapidMind `Program`, that can be called as a subroutine:

```
Program vector_mult = BEGIN {
   In<Value1f>a;
   In<Value1f>b;
   a = a*b;
}
```

where `Value1f` indicates that a and b are each floats (`Value4f` would indicate a vector of four floats). Next, `vector_mult` replaces the *for* loop, and `a[]` and `b[]` are defined with RapidMind types.

```
main {
   Array<1,Value1f> A(N);
   Array<1,Value1f> B(N);
   // ... initialize values ...
   A = vector_mult(A,B);
}
```

RapidMind automatically parallelizes and distributes the program over the target platform, hiding platform specific thread management and data transfer operations from the programmer. RapidMind also supports reduction functions, multi-dimensional `Array` types, and data views such as shifting or striping for manipulating the arrays.

4. BENCHMARK APPLICATIONS

For our experimental analysis we selected three benchmark applications that exemplify different spaces in the spectrum of applications which ranges from computation-bound applications to data-bound applications.

4.1 Monte Carlo for Option Pricing

An option is a right to sell or buy a financial asset at a predetermined price on a future date. An option itself can be traded and its price is the discounted profit made by exercising the option. The future price of the asset, which determines the present price of an option, can be predicted using the Black-Scholes option pricing model [12, 26]. This assumes that the price fluctuations of the underlying financial asset can be modeled as geometrical Brownian motion:

$$dS_t = \mu S_t dt + v S_t dW_t \tag{1}$$

where S_t is the asset price at time t, W_t is Wiener random process, and μ (drift) and v (volatility) are two constants. Based on Eq. 1, the asset price at time T is:

$$S_T = S_0 e^{((\mu - 0.5v^2)T + vT^{1/2}N(0,1))} \tag{2}$$

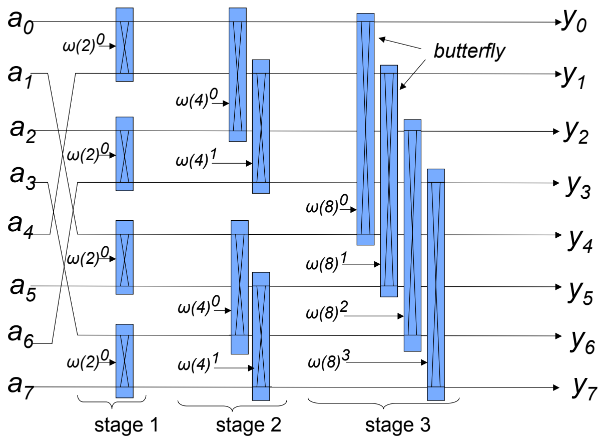

Figure 3: FFT computation on eight input signals.

where $N(0,1)$ is a normally-distributed random number between 0 and 1.

Based on Eq. 2, Monte Carlo simulations can be applied to estimate the expected value of S_T at a future time T. Each simulation consists of the following two steps:

1. generate a normally distributed random number x;
2. replace $N(0,1)$ in Eq. 2 with x to get one price S'_T.

The average of all the S'_T's obtained at step 2 is an estimation of the expected value of S_T in Eq. 2. For the estimation of the expected S_T to converge, a sufficient number of simulations must be performed. Since all simulations are independent from each other, they can be exercised by parallel processes between which there is little communication. Hence, this is an *embarrassingly parallel* workload that can be tackled by distributing the various simulations evenly across the processing elements of the multi-processor hardware platform. Such a workload represents one extreme on the spectrum of computation and communication patterns that we are considering in our experiments.

4.2 Fast-Fourier Transform (FFT)

Fast Fourier Transform (FFT) is a divide-and-conquer algorithm to compute the discrete-time Fourier transform (DFT), which converts discrete signals from time domain to frequency domain. Given an input of N discrete signals $(x_1, x_2, \ldots, x_N)$, the DFT $(X_1, X_2, \ldots, X_N)$ is defined as follows:

$$X_k = \sum_{n=0}^{N-1} x_n \, e^{-\frac{2\pi i}{N} nk}, \quad k = 0, \ldots, N-1$$

While a naïve implementation of this convolution requires $O(n^2)$ floating point operations, the FFT requires only $O(n \log n)$ of them.

Figure 3 illustrates a simple implementation of FFT on eight signals. The input signals are fed from the left side, and the transformed signals are output from the right. The FFT is divided into three stages, and each stage has four "butterfly" operations, each of which is applied to two signals (this is the so called radix-2 FFT). Since the butterfly

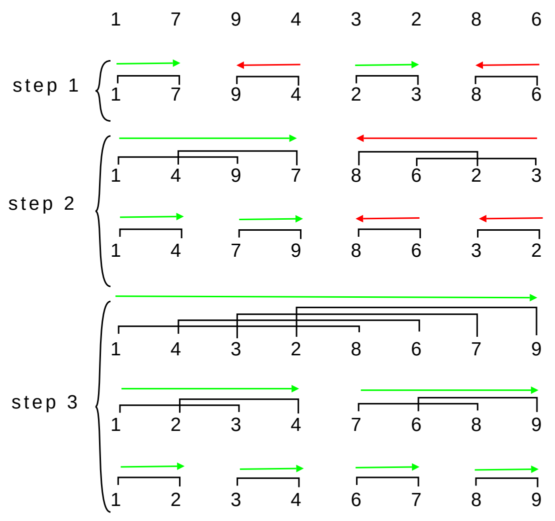

Figure 4: Structure of Bitonic Sort Algorithm

operations of a single stage are independent, they can be executed in parallel. The FFT belongs to the class of *spectral methods* [11].

The FFT benchmark requires both intensive computational and communication support from the hardware execution platform. From the computational aspect, a butterfly operation takes in two inputs and a complex number $\omega(k)$, called "twiddle factor", which is the k-th root of unity, and performs two sets of floating point additions and one set of multiplications. To compute each twiddle factor, two trigonometric function calls are required.

The communication aspect of FFT is also very challenging. Each butterfly box reads two input signals, which may reside in remote memory (like in the local store of another SPE on Cell) or may be stored in the same bank of memory (like in the same bank of the global memory on G80). In the formal case, inter-process communications (like DMAs on Cell) are required to bring in the right data before butterfly operations can be applied; in the latter case the memory bandwidth will degrade due to bank conflict.

4.3 Bitonic Sort

Bitonic sort is a popular $O(n\ log^2 n)$ sorting algorithm for parallel architectures. Although its complexity is less optimal than $O(n\ log\ n)$ sorting algorithms like merge sort, bitonic sort is desirable because the order of its compare-and-swap operations is not dependent on their outcome.

The bitonic sort algorithm uses a *divide-and-conquer* approach. To sort a list of elements, the list is broken into two even pieces. The two pieces are sorted in opposite directions and then merged together with the $(O(n))$ bitonic merge operation. Bitonic merge works as follows: assuming there are two lists sorted in opposite directions, first compare-and-swap the 1^{st} element to the $\frac{n}{2}^{th}$ element, next compare and swap the 2^{nd} element to the $(\frac{n}{2}+1)^{th}$ element, etc. Figure 4 shows how a list of 8 integers is sorted with the bitonic sort algorithm. In Step 1, every other set of two elements is sorted in ascending order, and the other sets are

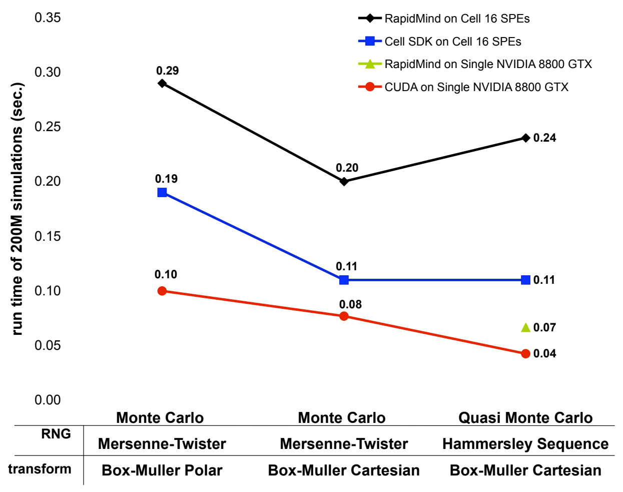

Figure 5: Performance comparison of Monte Carlo simulations for Black-Scholes option pricing.

sorted in descending order. In Step 2, every other set of four elements is sorted in ascending order, and so on.

Sorting, in general, is a task that requires very little computation, but intensive data movement. Bitonic sort can be implemented in a variety of ways. A recursive implementation is intuitive from the divide-and-conquer definition of the algorithm; however, non-recursive iterative implementations are typically used.

5. EXPERIMENTAL RESULTS

We first report the experimental results for each of the three benchmarks separately and then we present some preliminary conclusions that can be drawn by comparing them.

5.1 Option Pricing Experiments

We evaluate three distinct implementations of Monte Carlo simulations for option pricing with the Black-Scholes model. The three implementations compute the same pricing of an option, but differ in the way of generating and transforming random numbers. The first two approaches adopt the Mersenne-Twister random number generator [23], but use different methods for transforming uniformly distributed random numbers to normally distributed ones. The third approach uses Hammersley sequence, a low discrepancy sequence, instead of pseudo-random numbers[1].

Figure 5 reports the run time of each implementation for two hundred million simulations on both the Cell blade and the G80. Comparing the performance of hardware using platform-specific SDKs, G80 (using CUDA) outperforms the Cell blade (using Cell SDK) in all three variations of Monte Carlo simulations by a wide margin. Note that using Hammersley sequence on GPU boosts the performance significantly, compared to performances of the other two approaches based on pseudo random numbers. This is because the low-latency texture memory of GPUs can be exploited to store the read-only lookup table required by the Hammersley sequence algorithm. However, using Hammersley sequence

[1]This is referred as "quasi Monte Carlo".

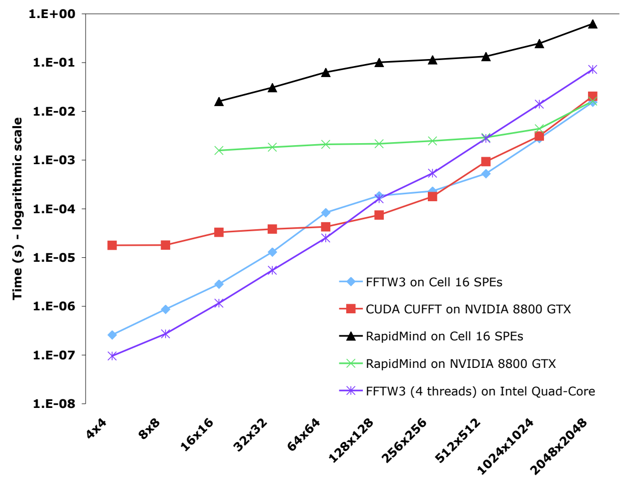

Figure 6: Performance comparison of single-precision 2-D FFT on input arrays of various sizes.

is no more advantageous on the Cell processor, which does not provide such specialized hardware.

Comparing the performance of RapidMind and platform-specific SDKs on the same hardware, the SDK versions run faster than their corresponding RapidMind versions both on Cell and on G80. Note that on G80, however, the RapidMind implementation of Mersenne-Twister random number generator cannot run on GPUs, because the algorithm reads and writes a local array, which is not supported by the latest RapidMind backend (version 2.1) of GPUs[2].

Both of the processing cores of Cell BE and G80 are SIMD-like architectures which favor straight lines of code. The usual control structures in most high-level programming languages, like `while`, `if`, `for` statements, can significantly degrade the performance of such architectures, since these control structures make the code hard to SIMDize. In particular, on both hardware platforms the Box-Muller polar transformation runs slower than the Cartesian version. The Box-Muller transformation requires less computation but relies on control structures (like a `while` loop); the Cartesian transformation use straight lines of code but needs two trigonometric function calls.

5.2 FFT Experiments

Figure 6 reports the run time of two-dimensional FFT on various architectures. On Cell we use one of the most popular FFT library, FFTW [14], whose core computation optimally combines several straight lines of code fragments called codelets written in platform natives; on G80 we use CUFFT [8], which is the FFT library in CUDA and provides a similar interface to FFTW. We also compare implementations using RapidMind on Cell and G80. The run time measurements are performed by interfacing the above libraries with the "benchFFT" environment, an extensible FFT benchmark program [1].

Depending on input sizes, the fastest FFT-performing architecture varies accordingly. For inputs smaller than 64×64 2D arrays, the FFTW library on Cell runs faster than the CUFFT on G80. For the inputs of size between (64×64 to 256×256), CUFFT/G80 outperforms FFTW/Cell. For input size beyond 256×256, the FFTW/Cell is slightly faster than CUFFT/G80.

The results of the FFT performance can be analyzed in the context of the computation and communication aspect of the FFT algorithm. The FFT algorithm requires not only intensive floating-point computations but also frequent data communications between processing elements. On most 2D FFT instances Cell's flexible, on-chip communication fabric overcomes its floating-point computation disadvantage with respect to G80, which does not provide direct links for inter-multiprocessor communications. Therefore for large inputs, the Cell edges G80, even though it has less floating-point computation capability.

The RapidMind implementations have worse performance compared to their SDK counterparts on both platforms. Compared to FFTW/Cell BE, this is not surprising because RapidMind's programming model does not support direct communications between concurrent processes, thus the powerful Cell on-chip ring is not utilized. On the other hand, RapidMind's limited communication model is actually based on GPU's hardware. Therefore on large data inputs RapidMind's performance is comparable to CUFFT/G80.

We also benchmark the FFT run time on a general-purpose Intel CPU (Intel Kentsfield quad-core clocked at 2.6GHz), and its results are also shown in Figure 6. For input data smaller than 128×128, the quad-core CPU has better FFT performance than Cell and G80, which incur the overhead of distributing the work to processing cores, including the time investment of "forking" and "joining" parallel processes on the processing units.

5.3 Bitonic Sort Experiments

Our RapidMind implementation of bitonic sort is an iterative loop-based solution, where the innermost loops are replaced with data-parallel RapidMind program calls. Our initial RapidMind implementation was tuned for the two chosen hardware execution platforms: we use vectorization in both cases and RapidMind local arrays (arrays within RapidMind programs) on the Cell in order to take advantage of the SPU local storage.

We compare our RapidMind implementation with CellSort [15] on the Cell QS20 and GPUSort [17] on the G80. The results for the different software implementations and hardware platforms are shown in Figure 7. The curves end at different input data sizes because the different implementations do not support the same maximum sizes. The RapidMind and CellSort implementations sorted integers, while the GPUSort implementation sorted floats (GPUs tend to handle floats more efficiently, but Cell handles both equally).

The performance of RapidMind bitonic sort comes much closer to the performance of hand-written code on the GPU than it does on the Cell. For smaller cases (<256K elements), RapidMind performs better than GPUSort. In contrast, RapidMind is on average a factor of 50 times slower on the Cell than CellSort.

The results show that the Cell blade is faster than G80 for computing the bitonic sort, but the gap diminishes as the input data size increases. Sorting 32K elements on the G80 is about 25 times slower, but sorting 8M integers is only 4 times slower. The curve of CellSort's performance has three distinct sections: 4K-32K, 32K-512K, and >512K. In the

[2]This restriction is lifted in RapidMind's Cell backend.

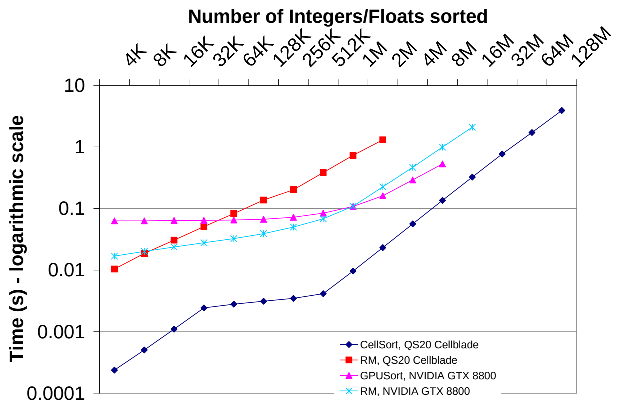

Figure 7: Bitonic sort results for various input sizes.

first section, the data is small enough to fit into the local store of a single SPU, and sorting is handled locally. In the second section, the data is too large for a single SPU's local store, but small enough to fit into the combined local stores so off-chip communication is not necessary during the sorting (except in the case of 512K where data must be transferred between the two Cell chips). In the last case, the problem size is too large to fit onto the chips and so data must be swapped in and out of main memory throughout the sort. In these larger problem sizes, the G80 begins to catch up because it has higher off-chip memory bandwidth than the Cell as shown in Figures 1 and 2.

5.4 Discussion

In this section we summarize our experiment results. In particular, we qualitatively evaluate the impact of the computation and communication aspects of the two hardware platforms and the three benchmark programs.

Figure 8 highlights the relative performance of the Cell blade (16 SPEs) and the G80 using platform-specific SDKs across the spectrum of different computation and communication patterns. If the ratio is above 1, the Cell blade is faster. The general trend is that G80 has an edge on computation bound workloads; in contrast Cell performs better on communication intensive applications. For example, the G80 runs faster if the application is computation bound, like Monte Carlo methods. On the other hand, Cell is faster than G80 on applications like FFT on large data inputs and bitonic sort, both of which require intensive data communications.

In particular, data movement is the limiting factor in the performance of bitonic sort. Thus the memory capacity and communication network of a multi-core play an important role for this application. The Cell's EIB gives the Cell a great advantage since cores can transfer data between each other very quickly. However as the problem size scales up and data must be swapped in and out of the off-chip memory, the bandwidth to main memory has more of an impact.

The relative performance of RapidMind programs and their platform-specific SDK counterparts are reported in Figure 9. If a RapidMind program runs faster, its relative ratio is larger than one. Except for the bitonic sort OpenGL implementations on G80, RapidMind's performance cannot compete with the SDK-based implementations yet. At best RapidMind program are slightly faster (only in one problem

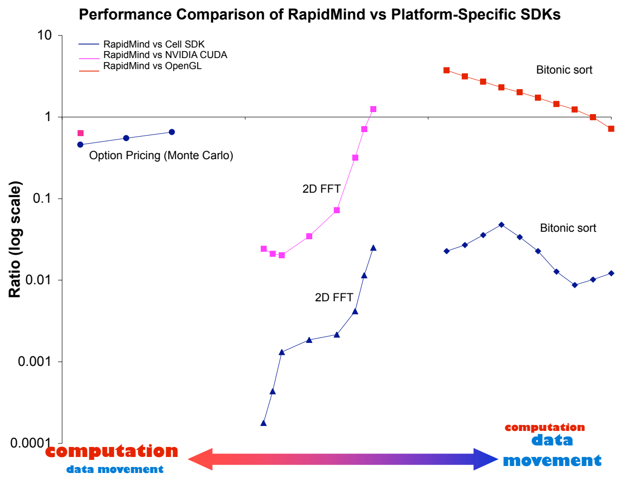

Figure 8: Performance comparison of Cell and NVIDIA 8800 GTX. The x-axis shows the spectrum of computation and communication patterns. Data points toward the left side are more computation bound; data points on the right are more communication bound.

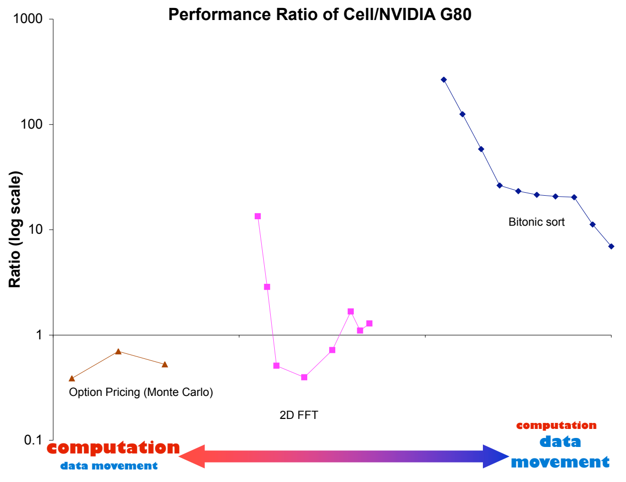

Figure 9: Performance comparison of RapidMind and platform-specific SDKs.

instance), but at worst they are orders of magnitude slower than the corresponding SDK versions.

One reason of this performance gap is due to RapidMind's programming model, which is MapReduce. This fits applications like Monte Carlo methods well, which are special cases of the MapReduce pattern. However, this programming model provides no inter-process communications. This is overly restrictive for applications like FFT and sorting where data movements between parallel processes are as important as computation itself. Especially when these applications are implemented on Cell using RapidMind, the high-bandwidth inter-SPE communication links are not exercised, thus leading to poor overall performance.

6. CONCLUSIONS

We evaluate the performance of two leading multicore architectures: IBM Cell processor and NVIDIA GeForce 8800 GTX as a GPU. We find that the Cell BE and GeForce 8800 GTX excel over different domains of general-purpose applications. The differences in the types of applications that perform best with each architecture reflect the differences in the communication and computation strengths of the architectures. The Cell allows high-bandwidth communication between SPEs; accordingly, the Cell outperforms the GeForce 8800 GTX for communication-bound applications (like FFT and bitonic sort). On the other hand, while the GeForce 8800 GTX uses a shared memory approach and does not have a mechanism for direct communication, it has its strength in the large number of processing units. With the ability to perform four times more floating point operations concurrently than the Cell, the GeForce 8800 GTX performs better in applications which are computation-bound like Monte Carlo simulations.

In addition to comparing which architectures are better suited to general purpose computation, we also study what tools are most effective over a broad class of applications. The new multicore programming tool RapidMind provides an abstract programming interface for multicore systems and frees the programmer from managing low-level thread and memory management in parallel programs. However, despite its elegant programming interface, RapidMind struggles to deliver performance for some applications. The greatest performance losses are experienced with the Cell architecture for applications that require non-trivial data movement. We conclude that inter-core communication is an important aspect of multicore architectures for the overall class of general-purpose applications, and we expect that its importance will grow as the number of processing units per chip increases in the future. Not only must multicore programming tools fully utilize a multicore system's communication capabilities, but the multicore architectures must also be designed with programmable communication networks.

7. REFERENCES

[1] benchFFT. [Online]. Available: `http://www.fftw.org/benchfft/`.

[2] GPGPU - General-Purpose Computation Using Graphics Hardware. [Online]. Available: `http://www.gpgpu.org/`.

[3] IBM Corporation Press Release, *IBM to Build World's First Cell Broadband Engine Based Supercomputer*. [Online]. Available: `http://www-304.ibm.com/jct03004c/press/us/en/pressrelease/20210.wss`.

[4] IBM Corporation, Software Development Kit 2.1 Accelerated Library Framework Programmer's Guide and API Reference Version 1.1.

[5] Mercury Computer Systems. *25U/42U Dual Cell-Based Blade 2 System*. [Online]. Available: `http://www.mc.com/microsites/cell/`.

[6] NVIDIA CUDA Toolkit. [Online]. Available: `http://www.nvidia.com/cuda`.

[7] NVIDIA Tesla GPU computing solutions for HPC. [Online]. Available: `http://www.nvidia.com/tesla`.

[8] CUDA CUFFT library, June 2007. [Online]. Available: `http://developer.download.nvidia.com/compute/cuda/1_1/CUFFT_Library_1.1.pdf`.

[9] NVIDIA GeForce 8800 GTX, 2007. [Online]. Available: `http://www.nvidia.com/page/8800_tech_briefs.html`.

[10] T. W. Ainsworth and T. M. Pinkston. Characterizing the Cell EIB on-chip network. *IEEE Micro*, 27(5):6–14, 2007.

[11] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelik. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California at Berkeley, December 2006.

[12] F. Black and M. Scholes. The pricing of options and corporate liabilities. *Journal of Political Economy*, 81(3):637–654, 1973.

[13] J. Easton, I. Meents, O. Stephan, H. Zisgen, and S. Kato. Porting financial markets applications to the cell broadband engine architecture. [Online]. Available: `http://dl.alphaworks.ibm.com/technologies/cellsw/cellFMwhitepaper.pdf`.

[14] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[15] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: High performance sorting on the Cell processor. In *Very Large Data Bases Conference (VLDB)*, Vienna, Austria, September 2007.

[16] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGOPS Oper. Syst. Rev.*, 40(5):151–162, 2006.

[17] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *ACM SIGMOD*, 2006.

[18] M. Gschwind, D. Erb, S. Manning, and M. Nutter. An open source environment for Cell broadband engine system software. *IEEE Computer*, 40(6):37–47, 2007.

[19] C. R. Johns and D. A. Brokenshire. Introduction to the Cell broadband engine architecture. *IBM J. Res. Develop.*, 51(5):521–528, Sept. 2007.

[20] J. A. Kahle, M. N. Day, H. P. Hofsteee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the CELL multiprocessor. *IBM J. Res. Develop.*, 49(4/5):589–604, Sept. 2005.

[21] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[22] D. Luebke and G. Humphreys. How GPUs Work. *IEEE Computer*, 40(2):96–100, Feb. 2007.

[23] M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.

[24] M. McCool, S. D. Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 787–795, New York, NY, USA, 2004. ACM.

[25] M. D. McCool. Data-parallel programming on the

CELL BE and the GPU using the rapidmind development platform. *GSPx Multicore Applications Conference*, 2006.

[26] R. C. Merton. Theory of rational option pricing. *Bell Journal of Economics and Management Science*, 4(1):141–183, 1973.

[27] M. Monteyne. Rapidmind multi-core development platform. White paper, RapidMind Inc., November 2007.

[28] A. K. Nanda, J. R. Moulic, R. E. Hanson, G. Goldrian, M. N. Day, B. D. D'Amora, and S. Kesavarapu. Cell/B.E. blades: Building blocks for scalable, real-time, interactive, and digital media servers. *IBM J. Res. Develop.*, 51(5):573–782, Sept. 2007.

[29] J. Owens. Streaming architectures and technology trends. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 9, New York, NY, USA, 2005. ACM.

[30] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃijger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, Sept. 2005.

[31] D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *ISSCC*, pages 184–185, Feb. 2005.