Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU

Victor W Lee†, Changkyu Kim†, Jatin Chhugani†, Michael Deisher†, Daehyun Kim†, Anthony D. Nguyen†, Nadathur Satish†, Mikhail Smelyanskiy†, Srinivas Chennupaty⋆, Per Hammarlund⋆, Ronak Singhal⋆ and Pradeep Dubey†

victor.w.lee@intel.com

†Throughput Computing Lab, Intel Corporation

⋆Intel Architecture Group, Intel Corporation

ABSTRACT

Recent advances in computing have led to an explosion in the amount of data being generated. Processing the ever-growing data in a timely manner has made throughput computing an important aspect for emerging applications. Our analysis of a set of important throughput computing kernels shows that there is an ample amount of parallelism in these kernels which makes them suitable for today's multi-core CPUs and GPUs. In the past few years there have been many studies claiming GPUs deliver substantial speedups (between 10X and 1000X) over multi-core CPUs on these kernels. To understand where such large performance difference comes from, we perform a rigorous performance analysis and find that after applying optimizations appropriate for both CPUs and GPUs the performance gap between an Nvidia GTX280 processor and the Intel Core i7 960 processor narrows to only 2.5x on average. In this paper, we discuss optimization techniques for both CPU and GPU, analyze what architecture features contributed to performance differences between the two architectures, and recommend a set of architectural features which provide significant improvement in architectural efficiency for throughput kernels.

Categories and Subject Descriptors

C.1.4 [**Processor Architecture**]: Parallel architectures
; C.4 [**Performance of Systems**]: Design studies
; D.3.4 [**Software**]: Processors—*Optimization*

General Terms

Design, Measurement, Performance

Keywords

CPU architecture, GPU architecture, Performance analysis, Performance measurement, Software optimization, Throughput Computing

1. INTRODUCTION

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISCA'10, June 19–23, 2010, Saint-Malo, France.
Copyright 2010 ACM 978-1-4503-0053-7/10/06 ...$10.00.

The past decade has seen a huge increase in digital content as more documents are being created in digital form than ever before. Moreover, the web has become the medium of choice for storing and delivering information such as stock market data, personal records, and news. Soon, the amount of digital data will exceed exabytes (10^{18}) [31]. The massive amount of data makes storing, cataloging, processing, and retrieving information challenging. A new class of applications has emerged across different domains such as database, games, video, and finance that can process this huge amount of data to distill and deliver appropriate content to users. A distinguishing feature of these applications is that they have plenty of data level parallelism and the data can be processed independently and in any order on different processing elements for a similar set of operations such as filtering, aggregating, ranking, etc. This feature together with a processing deadline defines *throughput computing applications*. Going forward, as digital data continues to grow rapidly, throughput computing applications are essential in delivering appropriate content to users in a reasonable duration of time.

Two major computing platforms are deemed suitable for this new class of applications. The first one is the general-purpose CPU (central processing unit) that is capable of running many types of applications and has recently provided multiple cores to process data in parallel. The second one is the GPU (graphics processing unit) that is designed for graphics processing with many small processing elements. The massive processing capability of GPU allures some programmers to start exploring general purpose computing with GPU. This gives rise to the GPGPU field [3, 33].

Fundamentally, CPUs and GPUs are built based on very different philosophies. CPUs are designed for a wide variety of applications and to provide fast response times to a single task. Architectural advances such as branch prediction, out-of-order execution, and super-scalar (in addition to frequency scaling) have been responsible for performance improvement. However, these advances come at the price of increasing complexity/area and power consumption. As a result, main stream CPUs today can pack only a small number of processing cores on the same die to stay within the power and thermal envelopes. GPUs on the other hand are built specifically for rendering and other graphics applications that have a large degree of data parallelism (each pixel on the screen can be processed independently). Graphics applications are also latency tolerant (the processing of each pixel can be delayed as long as frames are processed at interactive rates). As a result, GPUs can trade off single-thread performance for increased parallel processing. For instance, GPUs can switch from processing one pixel to another when long

latency events such as memory accesses are encountered and can switch back to the former pixel at a later time. This approach works well when there is ample data-level parallelism. The speedup of an application on GPUs is ultimately limited by the percentage of the scalar section (in accordance with Amdahl's law).

One interesting question is the relative suitability of CPU or GPU for throughput computing workloads. CPUs have been the main workhorse for traditional workloads and would be expected to do well for throughput computing workloads. There is little doubt that today's CPUs would provide the best single thread performance for throughput computing workloads. However, the limited number of cores in today's CPUs limits how many pieces of data can be processed simultaneously. On the other hand, GPUs provide many parallel processing units which are ideal for throughput computing. However, the design for graphics pipeline lacks some critical processing capabilities (e.g., large caches) for general purpose workloads, which may result in lower architecture efficiency on throughput computing workloads.

This paper attempts to correlate throughput computing characteristics with architectural features on today's CPUs and GPUs and provides insights into why certain throughput computing kernels perform better on CPUs and others work better on GPUs. We use a set of kernels and applications that have been identified by previous studies [6, 10, 13, 44] as important components of throughput computing workloads. We highlight the importance of platform-specific software optimizations, and recommend an application-driven design methodology that identifies essential hardware architecture features based on application characteristics.

This paper makes the following contributions:

- We reexamine a number of claims [9, 19, 21, 32, 42, 45, 47, 53] that GPUs perform 10X to 1000X better than CPUs on a number of throughput kernels/applications. After tuning the code for **BOTH** CPU and GPU, we find the GPU only performs 2.5X better than CPU. This puts CPU and GPU roughly in the same performance ballpark for throughput computing.
- We provide a systematic characterization of throughput computing kernels regarding the types of parallelism available, the compute and bandwidth requirements, the access pattern and the synchronization needs. We identify the important software optimization techniques for efficient utilization of CPU and GPU platforms.
- We analyze the performance difference between CPU and GPU and identify the key architecture features that benefit throughput computing workloads.

This paper is organized as follows: Section 2 discusses the throughput computing workloads used for this study. Section 3 describes the two main compute platforms – CPUs and GPUs. Section 4 discusses the performance of our throughput computing workloads on today's compute platforms. Section 5 provides a platform-specific optimization guide and recommends a set of essential architecture features. Section 6 discusses related work and Section 7 concludes our findings.

2. THE WORKLOAD: THROUGHPUT COMPUTING KERNELS

We analyzed the core computation and memory characteristics of recently proposed benchmark suites [6, 10, 13, 44] and formulated the set of *throughput computing kernels* that capture these characteristics. These kernels have a large amount of data-level parallelism, which makes them a natural fit for modern multi-core architectures. Table 1 summarizes the workload characterization. We classify these kernels based on (1) their compute and memory requirements, (2) regularity of memory accesses, which determines the ease of exploiting data-level parallelism (SIMD), and (3) the granularity of tasks, which determines the impact of synchronization. These characteristics provide insights into the architectural features that are required to achieve good performance.

1. SGEMM (both dense and sparse) is an important kernel that is an integral part of many linear algebra numerical algorithms, such as linear solvers. SGEMM is characterized by regular access patterns and therefore maps to SIMD architecture in a straightforward manner. Threading is also simple, as matrices can be broken into sub-blocks of equal size which can be operated on independently by multiple threads. SGEMM performs $O(n^3)$ compute, where n is the matrix dimension and has $O(n^2)$ data accesses. The ratio of compute to data accesses is $O(n)$, which makes SGEMM a compute-bound application, when properly blocked.

2. MC or Monte Carlo randomly samples a complex function, with an unknown or highly complex analytical representation, and averages the results. We use an example of Monte Carlo from computational finance for pricing options [34]. It simulates a random path of an underlying stock over time and calculates a payoff from the option at the end of the time step. It repeats this step many times to collect a large number of samples which are then averaged to obtain the option price. Monte Carlo algorithms are generally compute-bound with regular access patterns, which makes it a very good fit for SIMD architectures.

3. Conv or convolution is a common image filtering operation used for effects such as blur, emboss and sharpen. Its arithmetic computations are simple multiply-add operations and its memory accesses are regular in small neighborhood. Each pixel is calculated independently, thus providing ample parallelism at both SIMD and thread level. Though its compute-to-memory characteristic varies depending on the filter size, in practice, it usually exhibits high compute-to-memory ratio. Its sliding-window-style access pattern gives rise to a memory alignment issue in SIMD computations. Also, multi-dimensional convolutions incur non-sequential data accesses, which require good cache blocking for high performance.

4. FFT or Fast Fourier Transform is one of the most important building blocks for signal processing applications. It converts signals from time domain to frequency domain, and vice versa. FFT is an improved algorithm to implement Discrete Fourier Transform (DFT). DFT requires $O(n^2)$ operations and FFT improves it to $O(n \log n)$. FFT algorithms have been studied exhaustively [26]. Though various optimizations have been developed for each usage model/hardware platform, their basic behavior is similar. It is composed of $\log n$ stages of the butterfly computation followed by a bit-reverse permutation. Arithmetic computations are simple floating-point multiply-adds, but data access patterns are non-trivial pseudo-all-to-all communication, which makes parallelization and SIMDification difficult. Therefore, many studies [7] have focused on the challenges to implement FFT on multi-core wide-SIMD architectures well.

5. SAXPY or Scalar Alpha X Plus Y is one of the functions in the Basic Linear Algebra Subprograms (BLAS) package and is a combination of scalar multiplication and vector addition. It has a regular access pattern and maps well to SIMD. The use of TLP requires only a simple partitioning of the vector. For long vectors that do not fit into the on-die storage, SAXPY is bandwidth bound.

Kernel	Application	SIMD	TLP	Characteristics
SGEMM **(SGEMM) [48]**	Linear algebra	Regular	Across 2D Tiles	Compute bound after tiling
Monte Carlo **(MC) [34, 9]**	Computational Finance	Regular	Across paths	Compute bound
Convolution **(Conv) [16, 19]**	Image Analysis	Regular	Across pixels	Compute bound; BW bound for small filters
FFT **(FFT) [17, 21]**	Signal Processing	Regular	Across smaller FFTs	Compute/BW bound depending on size
SAXPY **(SAXPY) [46]**	Dot Product	Regular	Across vector	BW bound for large vectors
LBM **(LBM) [32, 45]**	Time Migration	Regular	Across cells	BW bound
Constraint Solver **(Solv) [14]**	Rigid body physics	Gather/Scatter	Across constraints	Synchronization bound
SpMV **(SpMV) [50, 8, 47]**	Sparse Solver	Gather	Across non-zero	BW bound for typical large matrices
GJK **(GJK) [38]**	Collision Detection	Gather/Scatter	Across objects	Compute Bound
Sort **(Sort) [15, 39, 40]**	Database	Gather/Scatter	Across elements	Compute bound
Ray Casting **(RC) [43]**	Volume Rendering	Gather	Across rays	4-8MB first level working set, over 500MB last level working set
Search **(Search) [27]**	Database	Gather/Scatter	Across queries	Compute bound for small tree, BW bound at bottom of tree for large tree
Histogram **(Hist) [53]**	Image Analysis	Requires conflict detection	Across pixels	Reduction/synchronization bound
Bilateral **(Bilat) [52]**	Image Analysis	Regular	Across pixels	Compute Bound

Table 1: Throughput computing kernels characteristics. The referred papers contains the best previous reported performance numbers on CPU/GPU platforms. Our optimized performance numbers are at least on par or better than those numbers.

For very short vectors, SAXPY spends a large portion of time performing horizontal reduction operation.

6. LBM or Lattice Boltzmann method, is a class of computational fluid dynamics. LBM uses the discrete Boltzmann equation to simulate the flow of a Newtonian fluid instead of solving the Navier Stokes equations. In each time step, for a D3Q19 lattice, LBM traverses the entire 3D fluid lattice and for each cell computes new distribution function values from the cell's 19 neighbors (including self). Within each time step, the lattice can be traversed in any order as values from the neighbors are computed from the previous time step. This aspect makes LBM suitable for both TLP and DLP. LBM has $O(n)$ compute and requires $O(n)$ data, where n is the number of cells. The working set consists of the data of the cell and its 19 neighbors. The reuse of these values is substantially less than convolution. Large caches do not improve the performance significantly. The lack of reuse also means that the compute to bandwidth ratio is low; LBM is usually bandwidth bound.

7. Solv or constraint solver is a key part of game physics simulators. During the execution of the physical simulation pipeline, a collision detection phase computes pairs of colliding bodies, which are then used as inputs to a constraint solving phase. The constraint solver operates on these pairs and computes the separating contact forces, which keeps the bodies from inter-penetrating into one another. The constraints are typically divided into batches of independent constraints [14]. SIMD and TLP are both exploited among independent constraints. Exploiting SIMD parallelism is however challenging due to the presence of gather/scatter operations required to gather/scatter object data (position, velocity) for different objects. Ideally, the constraint solver should be bandwidth bound, because it iterates over all constraints in a given iteration and the number of constraints for realistic large scale destruction scenes exceeds the capacity of today's caches. However, practical implementations suffer from synchronization costs across sets of independent constraints, which limits performance on current architectures.

8. SpMV or sparse matrix vector multiplication is at the heart of many iterative solvers. There are several storage formats of sparse matrices, compressed row storage being the most common. Computation in this format is characterized by regular access patterns over non-zero elements and irregular access patterns over the vector, based on column index. When the matrix is large and does not fit into on-die storage, a well optimized kernel is usually bandwidth bound.

9. GJK is a commonly used algorithm for collision detection and resolution of convex objects in physically-based animations/simulations in virtual environments. A large fraction of the run-time is spent in computing support map, i.e., the furthest vertex of the object along a given direction. Scalar implementation of GJK is compute bound on current CPUs and GPUs and can exploit DLP to further speedup the run-time. The underlying SIMD is exploited by executing multiple instances of the kernel on different pairs of objects. This requires gathering the object data (vertices/edges) of multiple objects into a SIMD register to facilitate fast support map execution. Hence the run-time is dependent on support for an efficient gather instruction by the underlying hardware. There also exist techniques [38] that can compute support map by memoizing the object into a lookup table, and performing lookups into these tables at run-time. Although still requiring gathers, this lookup can be performed using texture mapping units available on GPUs to achieve further speedups.

10. Sort or radix sort is a multi-pass sorting algorithm used in many areas including databases. Each pass sorts one digit of the input at a time, from least to most significant. Each pass involves data rearrangement in the form of memory scatters. On CPUs, the best implementation foregoes the use of SIMD and implements a scatter oriented rearrangement within cache. On GPUs, where SIMD use is important, the algorithm is rewritten using a 1-bit sort primitive, called split [39]. The split based code, however, has more scalar operations than the buffer code (since it works on a single bit at a time). The overall efficiency of SIMD use relative to optimized scalar code is therefore not high even for split code. The number of bits considered per pass of radix sort depends on the size of the local storage. Increasing cache sizes will thus improve performance (each doubling of the cache size will increase the number of bits per pass by one). Overall, radix sort has $O(n)$ bandwidth and compute requirements (where n is the number of elements to be sorted), but is usually compute bound due to the inefficiency of SIMD use.

11. RC or Ray Casting is an important visual application, used to visualize 3D datasets, such as CT data used in medical imaging. High quality algorithms, known as ray casting, cast rays through the volume, performing compositing of each voxel into a corresponding pixel, based on voxel opacity and color. Tracing multiple rays using SIMD is challenging, because rays can access non-contiguous memory locations, resulting in incoherent and irregular memory accesses. Some ray casting implementations perform a decent amount of computation, for example, gradient shading. The first level working set due to adjacent rays accessing the same volume data is reasonably small. However, the last level working set

can be as large as the volume itself, which is several gigabytes of data.

12. Search or in-memory tree structured index search is a commonly used operation in various fields of computer science, especially databases. For CPUs, the performance depends on whether the trees can fit in cache or not. For small trees (tree sizes smaller than the last-level cache (LLC)), the search operation is compute bound, and can exploit the underlying SIMD to achieve speedups. However, for large trees (tree sizes larger than the LLC), the last few levels of the tree do not fit in the LLC, and hence the run-time for search is bound by the available memory bandwidth. As far as the GPUs are concerned, the available high-bandwidth exceeds the required bandwidth even for large trees, and the run-time is compute bound. The run-time of search is proportional to the tree depth on the GTX280.

13. Hist or histogram computation is an important image processing algorithm which hashes and aggregates pixels from the continuous stream of data into a smaller number of bins. While address computation is SIMD friendly, SIMDification of the aggregation, however, requires hardware support for conflict detection, currently not available on modern architectures. The access pattern is irregular and hence SIMD is hard to exploit. Generally, multi-threading of histogram requires atomic operation support. However, there are several parallel implementations of histogram which use privatization. Typically, private histograms can be made to fit into available on-die storage. However, the overhead of reducing the private histograms is high, which becomes a major bottleneck for highly parallel architectures.

14. Bilat or bilateral filter is a common non-linear filter used in image processing for edge-preserving smoothing operations. The core computation has a combination of a spatial and an intensity filter. The neighboring pixel values and positions are used to compute new pixel values. It has high computational requirements and the performance should scale linearly with increased flops. Typical image sizes are large; TLP/DLP can be exploited by dividing the pixels among threads and SIMD units. Furthermore, the bilateral filter involves transcendental operations like computing exponents, which can significantly benefit from fast math units.

3. TODAY'S HIGH PERFORMANCE COMPUTE PLATFORMS

In this section, we describe two popular high-performance compute platforms of today: (1) A CPU based platform with an Intel Core i7-960 processor; and (2) A GPU based platform with an Nvidia GTX280 graphics processor.

3.1 Architectural Details

First, we discuss the architectural details of the two architectures, and analyze the total compute, bandwidth and other architectural features available to facilitate throughput computing applications.

3.1.1 Intel Core i7 CPU

The Intel Core i7-960 CPU is the latest multi-threaded multi-core Intel-Architecture processor. It offers *four* cores on the same die running at a frequency of 3.2GHz. The Core i7 processor cores feature an out-of-order super-scalar microarchitecture, with newly added 2-way hyper-threading. In addition to scalar units, it also has 4-wide SIMD units that support a wide range of SIMD instructions [24]. Each core has a separate 32KB L1 for both instructions and data, and a 256KB unified L2 data cache. All four cores share an 8MB L3 data cache. The Core i7 processor also features an on-die memory controller that connects to three channels of DDR memory. Table 2 provides the peak single-precision and double-precision FLOPS for both scalar and SSE units, and also the peak bandwidth available per die.

3.1.2 Nvidia GTX280 GPU

The Nvidia GTX280 is composed of an array of multiprocessors (a.k.a. SM). Each SM has 8 scalar processing units running in lockstep[1], each at 1.3 GHz. The hardware SIMD structure is exposed to programmers through thread warps. To hide memory latency, GTX280 provides hardware multi-threading support that allows hundreds of thread contexts to be active simultaneously. To alleviate memory bandwidth, the card includes various on-chip memories – such as multi-ported software-controlled 16KB memory (referred to as *local shared buffer*) and small non-coherent read-only caches. The GTX280 also has special functional units like the texture sampling unit, and math units for fast transcendental operations. Table 2 depicts the peak FLOPS and the peak bandwidth of the GTX280[2].

3.2 Implications for Throughput Computing Applications

We now describe how the salient hardware features on the two architectures differ from each other, and their implications for throughput computing applications.

Processing Element Difference: The *CPU core* is designed to work well for a wide range of applications, including single-threaded applications. To improve single-thread performance, the CPU core employs out-of-order super-scalar architecture to exploit instruction level parallelism. Each CPU core supports scalar and SIMD operations, with multiple issue ports allowing for more than one operation to be issued per cycle. It also has a sophisticated branch predictor to reduce the impact of branch misprediction on performance. Therefore, the size and complexity of the CPU core limits the number of cores that can be integrated on the same die.

In comparison, the processing element for GPU *or SM* trades off fast single thread performance and clock speed for high throughput. Each SM is relatively simple. It consists of a single fetch unit and eight scalar units. Each instruction is fetched and executed in parallel on all eight scalar units over four cycles for 32 data elements (a.k.a. a *warp*). This keeps the area of each SM relatively small, and therefore more SMs can be packed per die, as compared to the number of CPU cores.

Cache size/Multi-threading: CPU provides caches and hardware prefetchers to help programmers manage data implicitly. The caches are transparent to the programmer, and capture the most frequently used data. If the working set of the application can fit into the on-die caches, the compute units are used more effectively. As a result, there has been a trend of increasing cache sizes in recent years. Hardware prefetchers provide additional help to reduce memory latency for streaming applications. Software prefetch instructions are also supported to potentially *reduce* the latency incurred with irregular memory accesses. In contrast, GPU provides for a large number of light-weight threads to hide memory latency. Each *SM* can support up to 32 concurrent warps per multi-processor. Since all the threads within a warp execute the same instruction, the warps are switched out upon issuing memory requests. To capture repeated access patterns to the same data, GTX280 provides for a few local storages (shared buffer, constant cache and texture cache). The size

[1]We view 8 scalar units as SIMD lanes, hence 8-element wide SIMD for GTX280.

[2]The peak single-precision SIMD Flops for GTX280 is 311.1 and increases to 933.1 by including fused multiply-add and a multiply operation which can be executed in SFU pipeline.

	Num. PE	Frequency (GHz)	Num. Transistors	BW (GB/sec)	SP SIMD width	DP SIMD width	Peak SP Scalar FLOPS (GFLOPS)	Peak SP SIMD Flops (GFLOPS)	Peak DP SIMD Flops (GFLOPS)
Core i7-960	4	3.2	0.7B	32	4	2	25.6	102.4	51.2
GTX280	30	1.3	1.4B	141	8	1	116.6	311.1/933.1	77.8

Table 2: Core i7 and GTX280 specifications. BW: local DRAM bandwidth, SP: Single-Precision Floating Point, DP: Double-Precision Floating Point.

of the local shared buffer is just 16KB, and much smaller than the cache sizes on CPUs.

Bandwidth Difference: Core i7 provides a peak external memory bandwidth of 32 GB/sec, while GTX280 provides a bandwidth of around 141 GB/sec. Although the ratio of peak bandwidth is pretty large (∼4.7X), the ratio of bytes per flop is comparatively smaller (∼1.6X) for applications not utilizing fused multiply add in the SFU.

Other Differences: CPUs provide for fast synchronization operations, something that is not efficiently implemented on GPUs. CPUs also provide for efficient in-register cross-lane SIMD operations, like general shuffle and swizzle instructions. On the other hand, such operations are emulated on GPUs by storing the data into the shared buffer, and loading it with the appropriate shuffle pattern. This incurs large overheads for some throughput computing applications. In contrast, GPUs provide support for gather/scatter instructions from memory, something that is not efficiently implemented on CPUs. Gather/Scatter operations are important to increase SIMD utilization for applications requiring access to non-contiguous regions of memory to be operated upon in a SIMD fashion. Furthermore, the availability of special function units like texture sampling unit and math units for fast transcendental helps speedup throughput computing applications that spend a substantial amount of time in these operations.

4. PERFORMANCE EVALUATIONS ON CORE I7 AND GTX280

This section evaluates the performance of the throughput computing kernels on the Core i7-960 and GTX280 processors and analyzes the measured results.

4.1 Methodology

We measured the performance of our kernels on (1) a 3.2GHz Core i7-960 processor running the SUSE Enterprise Server 11 operating system with 6GB of PC1333 DDR3 memory on an Intel DX58SO motherboard, and (2) a 1.3GHz GTX280 processor (an eVGA GeForce GTX280 card with 1GB GDDR3 memory) in the same Core i7 system with Nvidia driver version 19.180 and the CUDA 2.3 toolkit.

Since we are interested in comparing the CPU and the GPU architectures at the chip level to see if any specific architecture features are responsible for the performance difference, we did not include the data transfer time for GPU measurements. We assume the throughput computing kernels are executed in the middle of other computations that create data in GPU memory before the kernel execution and use data generated by the kernel in GPU memory. For applications that do not meet our assumption, transfer time can significantly degrade performance as reported by Datta in [16]. The GPU results as presented here are an upper bound of what will be seen in actual applications for these algorithms.

For both CPU and GPU performance measurements, we have optimized most of the kernels individually for each platform. For some of the kernels, we have used the best available implementation that already existed. Specifically, evaluations of **SGEMM**, **SpMV**, **FFT** and **MC** on GTX280 have been done using code from [1, 8, 2, 34], respectively. For the evaluations of **SGEMM**, **SpMV** and **FFT** on Core i7, we used Intel MKL 10.0. Table 3 shows the performance of throughput computing kernels on Core i7 and GTX280 processor with the appropriate performance metric shown in the caption. *To the best of our knowledge, our performance numbers are at least on par and often better than the best published data.* We typically find that the highest performance is achieved when multiple threads are used per core. For Core i7, the best performance comes from running 8 threads on 4 cores. For GTX280, while the maximum number of warps that can be executed on one GPU *SM* is 32, a judicious choice is required to balance the benefit of multithreading with the increased pressure on registers and on-chip memory resources. Kernels are often run with 4 to 8 warps per core for best GPU performance.

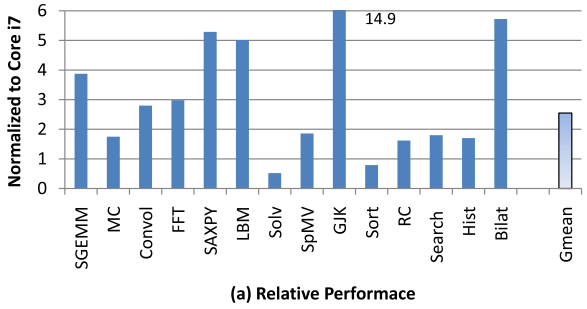

Figure 1: Comparison between Core i7 and GTX280 Performance.

4.2 Performance Comparison

Figure 1 shows the relative performance between GTX280 and Core i7 processors when data transfer time for GTX280 is not considered. Our data shows that GTX280 only has an average of 2.5X performance advantage over Core i7 in the 14 kernels tested. Only **GJK** achieves a greater than 10X performance gap due to the use of the texture sampler. **Sort** and **Solv** actually perform better on Core i7 . Our results are far less than previous claims like the 50X difference in pricing European options using Monte Carlo method [9], the 114X difference in LBM [45], the 40X difference in FFT [21], the 50X difference in sparse matrix vector multiplication [47] and the 40X difference in histogram computation [53], etc.

There are many factors that contributed to the big difference between previous reported results and ours. One factor is what CPU and GPU are used in the comparison. Comparing a high performance GPU to a mobile CPU is not an optimal comparison as their considerations for operating power, thermal envelop and reliability are totally different. Another factor is how much optimization is performed on the CPU and GPU. Many studies compare optimized GPU code to unoptimized CPU code and resulted in large difference. Other studies which perform careful optimizations to CPU and GPU such as [27, 39, 40, 43, 49] report much lower speedup similar to ours. Section 5.1 discusses the necessary software optimizations for improving performance for both CPU and GPU platforms.

Apps.	SGEMM	MC	Conv	FFT	SAXPY	LBM	Solv	SpMV	GJK	Sort	RC	Search	Hist	Bilat
Core i7-960	94	0.8	1250	71.4	16.8	85	103	4.9	67	250	5	50	1517	83
GTX280	364	1.4	3500	213	88.8	426	52	9.1	1020	198	8.1	90	2583	475

Table 3: Raw performance measured on the two platforms. From the left, metrics are Gflops/s, billion paths/s, million pixels/s, Gflops/s, GB/s, million lookups/s, FPS, Gflops/s, FPS, million elements/s, FPS, million queries/s, million pixels/s, million pixels/s.

4.3 Performance Analysis

In this section, we analyze the performance results and identify the architectural features that contribute to the performance of each of our kernels. We begin by first identifying kernels that are purely bounded by one of the two fundamental processor resources - bandwidth and compute. We then identify the role of other architectural features such as hardware support for irregular memory accesses, fast synchronization and hardware for performing fixed function computations (such as texture and transcendental math operations) in speeding up the remaining kernels.

4.3.1 Bandwidth

Every kernel requires some amount of external memory bandwidth to bring data into the processor. The impact of external memory bandwidth on kernel performance depends on two factors: (1) whether the kernel has enough computation to fully utilize the memory accesses; and (2) whether the kernel has a working set that fits in the on-die storages (either cache or buffers). Two of our kernels, **SAXPY** and **LBM** have large working sets that require global memory accesses without much compute on the loaded data - they are *purely bandwidth bound*. These kernels will benefit from increased bandwidth resources. The performance ratios for these two kernels between GTX280 and Core i7 are 5.3X and 5X respectively. These results are inline with the ratio between the two processors' peak memory bandwidth (which is 4.7X).

SpMV also has a large working set and very little compute. However, the performance ratio for this kernel between GTX280 and Core i7 is 1.9X, which is about 2.3X lower than the ratio of peak bandwidth between the two processors. This is due to the fact that the GTX280 implementation of SpMV keeps both vector and column index data structures in GDDR since they do not fit in the small on-chip shared buffer. However, in the Core i7 implementation, the vectors always fit in cache and the column index fits in cache for about half the matrixes. On average, the GPU bandwidth requirement for SpMV is about 2.5X the CPU requirement. As the result, although the GTX280 has 4.7X more bandwidth than Core i7 , the performance ratio is only 1.9X.

Our other kernels either have a high compute-to-bandwidth ratio or working sets that fit completely or partially in the on-die storage, thereby reducing the impact of memory bandwidth on performance. These categories of kernels will be described in later sections.

4.3.2 Compute Flops

The computational flops available on a processor depend on single-thread performance, as well as TLP due to the presence of multiple cores, or DLP due to wide vector (SIMD) units. While most applications (except the bandwidth-bound kernels) can benefit from improved single-thread performance and thread-level parallelism by exploiting additional cores, not all of them can exploit SIMD well. We identify **SGEMM**, **MC**, **Conv**, **FFT** and **Bilat** as being able to exploit all available flops on both CPU and GPU architectures. Figure 1 shows that **SGEMM**, **Conv** and **FFT** have GTX280-to-Core i7 performance ratios in the 2.8-4X range. This is close to the 3-6X single-precision (SP) flop ratio of the GTX280 to Core i7 architectures (see Table 2), depending on whether kernels can utilize fused multiply-adds or not.

The reason for not achieving the peak compute ratio is because GPUs do not achieve peak efficiency in the presence of shared buffer accesses. Volkov et al. [48] show that GPUs obtain only about 66% of the peak flops even for **SGEMM** (known to be compute bound). Our results match their achieved performance ratios. **MC** uses double precision arithmetic, and hence has a performance ratio of 1.8X, close to the 1.5X double-precision (DP) flop ratio. **Bilat** utilizes fast transcendental operations on GPUs (described later), and has a GTX280 to Core i7 performance ratio better than 5X. The algorithm used for **Sort** critically depends on the SIMD width of the processor. A typical radix sort implementation involves reordering data involving many scalar operations for buffer management and data scatters. However, scalar code is inefficient on GPUs, and hence the best GPU sort code uses a SIMD friendly split primitive. This has many more operations than the scalar code - and is consequently **1.25X slower** on the GTX280 than on Core i7.

Seven of our fourteen kernels have been identified as bounded by compute or bandwidth resources. We now describe the other architectural features that have a performance impact on the other seven kernels.

4.3.3 Cache

As mentioned in the section 4.3.1, on-die storage can alleviate external memory bandwidth pressure if all or part of the kernel's working set can fit in such storage. When the working set fits in cache, most kernels are compute bound and the performance will scale with increasing compute. The five kernels that we identify as compute bound have working sets that can be tuned to fit in any reasonably sized cache without significant loss of performance. Consequently, they only rely on the presence on some kind of on-chip storage and are compute bound on both CPUs and GPUs.

There are kernels whose working set cannot be easily tuned to any given cache size without loss of performance. One example is radix sort, which requires a working set that increases with the number of bits considered per pass of the sort. The number of passes over the data, and hence the overall runtime, decreases as we increase cache size. On GPUs with a small local buffer of 16 KB shared among many threads, we can only sort 4 bits in one pass - requiring 8 passes to sort 32-bit data. On Core i7, we can fit the working set of 8 bits in L2 cache; this only requires 4 passes - a 2X speedup. This contributes to **Sort** on Core i7 being **1.25X faster** than GTX280. Another interesting example is index tree search (**Search**). Here, the size of the input search tree determines the working set. For small trees that fit in cache, search on CPUs is compute bound, and in fact is **2X faster** than GPU search. For larger trees, search on CPUs is bandwidth bound, and becomes **1.8X slower** than GPUs. GPU search, in contrast, is always compute bound due to ISA inefficiencies (i.e., the unavailability of cross-lane SIMD operations).

Another important working set characteristic that determines kernel performance is whether the working set scales with the number of threads or is shared by all threads. Kernels like **SGEMM**, **MC**, **Conv**, **FFT**, **Sort**, **RC**, and **Hist** have working sets that scale with the number of threads. These kernels require larger working sets for GPUs (with more threads) than CPUs. This may not have any performance impact if the kernel can be tiled to fit into a cache of an

arbitrary size (e.g. **SGEMM** and **FFT**). However, tiling can only be done to an extent for **RC**. Consequently, **RC** becomes bandwidth bound on GPUs, which have very small amount of on-die storages, but is not bandwidth bound on CPUs (instead being affected by gathers/scatters, described later), and the performance ratio on GTX280 to Core i7 is only 1.6X, far less than bandwidth and compute ratios.

4.3.4 Gather/Scatter

Kernels that are not bandwidth bound can benefit with increasing DLP. However, the use of SIMD execution units places restrictions on kernel implementations, particularly in the layout of the data. Operands and results of SIMD operations are typically required to be grouped together sequentially in memory. To achieve the best performance, they should be placed into an address-aligned structure (for example for 4-wide single-precision SIMD, the best performance will be when the data is 16-byte aligned). If the data does not meet these layout restrictions, programmers must convert the data layout of kernels. This generally involves gather/scatter operations, where operands are gathered from multiple locations and packed together into a tight grouping, and results are scattered from a tight grouping to multiple locations. Performing gather/scatter in software can be expensive.[3] Thus, efficient hardware support for gather/scatter operations is very important.

A number of our kernels rely on the availability of gather/scatter operations. For example, **GJK** spends a large fraction of its runtime in computing the support map. This requires gathering the object data (vertices/edges) of multiple objects into a SIMD register to facilitate fast support map execution. Another example is **RC**, which requires gathering volume data across the rays. Frequent irregular memory accesses result in large number of gather operations. Up to 10% of the dynamic instructions are gather requests.

On Core i7, there is no hardware gather/scatter support. Consequently, **GJK** and **RC** do not utilize SIMD efficiently. For example, **RC** sees very incremental benefit from SSE between 0.8X and 1.2X, due to large overhead of software gather. **GJK** also sees minimal benefits from SSE. On GTX280, support for gather/scatter is offered for accesses to the local buffer and GDDR memory. Local shared buffer supports simultaneous gather/scatter accesses to multiple banks. The GDDR memory controller coalesces requests to the same line to reduce the number of gather/scatter accesses. This improved gather/scatter support leads to an improvement of **GJK** performance on the GTX280 over the Core i7 . However, gather/scatter support only has a small impact (of 1.2X) on **RC** performance because the accesses are widely spread out to memory, requiring multiple GDDR accesses even with coalescing support – it therefore becomes limited by GPU memory bandwidth. Consequently, the ratio of GTX280 to Core i7 performance for **RC** is only 1.6X, slightly better than the scalar flop ratio of 1.5X.

4.3.5 Reduction and Synchronization

Throughput computing kernels achieve high performance through thread-level (multiple cores and threads) and/or data-level (wide vector) parallelism. Reduction and synchronization are two operations that do not scale with increasing thread count and data-level parallelism. Various optimization techniques have been proposed to reduce the need of reduction and to avoid synchronization. However, the synchronization overhead is still dominant in some kernels such as **Hist** and **Solv**, and will become an even bigger performance bottleneck as the number of cores/threads and the SIMD width increase.

The performance of **Hist** is mainly limited by atomic updates. Although Core i7 supports a hardware lock increment instruction, 28% of the total run-time is still spent on atomic updates. Atomic update support on the GTX280 is also very limited. Consequently, a privatization approach where each thread generates a local histogram was implemented for both CPU and GPU. However, this implement does not scale with increase core count because the reduction overhead increases with the number of cores. Also, the lack of cross-SIMD lane operations like reduction on GPUs leads to large instruction overhead on GTX280. Thus, **Hist** is only 1.8X faster on GTX280 than on Core i7, much lower than the compute and bandwidth ratios ($\sim$5X). As was mentioned in Section 2, mapping **Hist** to SIMD requires support for conflict detection which is not currently available on modern architectures. Our analysis of ideal conflict detection hardware, capable of detecting an arbitrary number of conflicting indices within the same SIMD vector, improves histogram computation by up to 3X [29].

In **Solv**, a batch of independent constraints is executed simultaneously by multiple cores/threads, followed by a barrier before executing the next batch. Since resolving a constraint requires only small amount of computation (on the order of several hundred instructions), the task granularity is small. As a result, the execution time is dominated by the barrier overhead. On Core i7, barriers are implemented using atomic instructions. While it is possible to implement barrier operations entirely on GPU [48], this implementation does not guarantee that previous accesses to all levels of memory hierarchy have completed. CPUs provide a memory consistency model with the help of a cache coherence protocol. Due to the fact that cache coherence is not available on today's GPUs, assuring memory consistency between two batches of constraints requires launching the second batch from the CPU host, which incurs additional overhead. As a result, the barrier execution time of GTX280 is order of magnitude slower than on Core i7, resulting in an overall **1.9X slow down** in performance for GTX280 when compared to Core i7 for the constraint solver.

4.3.6 Fixed Function

Bilat consists of *transcendental operations* like computing exponential and power functions. However, for image processing purpose, high accuracy version of these functions are not necessary. Current CPUs use algebraic expressions to evaluate such expressions up to the required accuracy, while modern GPUs provide hardware to speedup the computation. On Core i7 , a large portion of run-time (around 66%) is spent in transcendental computation. On GTX280, due to the presence of fast transcendental hardware, it achieves a 5.7X performance ratio compare to Core i7 (much more than the peak compute ratio of around 3X). Speeding up transcendental on Core i7 (for example, as on GTX280) would improve **Bilat** performance by around 2X, and the resultant GPU-to-CPU performance ratio would be around 3X, which is closer to the peak compute ratio. **MC** is another kernel that would benefit from fast transcendental on CPUs.

Modern GPUs also provide for other fixed function units like the texture sampling unit, which is a major component of rendering algorithms. However, by reducing the linear-time support-map computation to constant-time texture lookups, **GJK** collision detection algorithm can exploit the **fast texture lookup** capability of GPUs, resulting in an overall 14.9X speedup on GTX280 over Core i7.

5. DISCUSSION

The platform-specific software optimization is critical to fully

[3]For 4-wide SIMD on Core i7, a compiler generated gather sequence will take 20 instructions and even a hand optimized assembly sequence will still take 13 instructions.

utilize compute/bandwidth resources for both CPUs and GPUs. We first discuss these software optimization techniques and derive a number of key hardware architecture features which play a major role in improving performance of throughput computing workloads.

5.1 Platform Optimization Guide

Traditionally, CPU programmers have heavily relied on increasing clock frequencies to improve performance and have not optimized their applications to fully extract TLP and DLP. However, CPUs are evolving to incorporate more cores with wider SIMD units, and it is critical for applications to be parallelized to exploit TLP and DLP. In the absence of such optimizations, CPU implementations are sub-optimal in performance and can be orders of magnitude off their attainable performance. For example, the previously reported **LBM** number on GPUs claims 114X speedup over CPUs [45]. However, we found that with careful multithreading, reorganization of memory access patterns, and SIMD optimizations, the performance on both CPUs and GPUs is limited by memory bandwidth and the gap is reduced to only 5X. Now we highlight the key platform-specific optimization techniques we learned from optimizing the throughput computing kernels.

CPU optimization: First, most of our kernels can linearly scale with the number of cores. Thus multithreading provides 3-4X performance improvement on Core i7. Second, CPUs heavily rely on caches to hide memory latency. Moreover, memory bandwidth on CPUs is low as compared to GPUs. Blocking is one technique which reduces LLC misses on CPUs. Programmers must be aware of the underlying cache hierarchy or use auto-tuning techniques to obtain the best performing kernels [18, 35]. Many of our kernels, **SGEMM**, **FFT**, **SpMV**, **Sort**, **Search**, and **RC** use cache blocking. **Sort**, for best performance, requires the number of bits per pass to be tuned so that its working set fits in cache. **RC** blocks the volume to increase 3D locality between rays in a bundle. We observe that cache blocking improves the performance of **Sort** and **Search** by 3-5X. Third, we found that reordering data to prevent irregular memory accesses is critical for SIMD utilization on CPUs. The main reason is that CPUs do not have gather/scatter support. **Search** performs explicit SIMD blocking to make memory accesses regular. **Solv** performs a reordering of the constraints to improve memory access patterns. Other kernels, such as **LBM** and **RC** convert some of the data structures from array-of-structure to structure-of-array format to completely eliminate gather operations. For example, the performance of **LBM** improves by 1.5X from this optimization.

GPU optimization: For GPUs, we found that global inter-thread synchronization is very costly, because it involves a kernel termination and new kernel call overhead from the host. **Hist** minimizes global synchronization by privatizing histograms. **Solv** also performs constraint reordering to minimize conflicts among neighboring constraints, which are global synchronization points. Another important optimization for GPUs was the use of the local shared buffer. Most of our kernels use the shared buffer to reduce bandwidth consumption. Additionally, our GPU sort uses the fact that buffer memory is multi-banked to enable efficient gathers/scatters of data.

5.2 Hardware Recommendations

In this section we capitalize on the learning from Section 4.3 to derive a number of key processor features which play major role in improving performance of throughput computing applications.

High compute flops and memory bandwidth: High compute flops can be achieved in two ways - by increasing core count or increasing SIMD width. While increasing core count provides higher performance benefits, it also incurs high area overhead. Increasing SIMD width also provides higher performance, and is more area-efficient. Our observation is confirmed by the trend of increasing SIMD width in computing platforms such as Intel architecture processor with AVX extension [23], Larrabee [41] and next generation Nvidia GT GPUs [30]. Increasing SIMD width will reach a point of diminishing return. As discussed in Section 4.3.4, irregular memory accesses can significantly decrease SIMD efficiency. The cost to fix this would offset any area benefit offered by increase SIMD width. Consequently, the future throughput computing processor should strike the right balance between SIMD and MIMD execution.

With the growth of compute flops, high memory bandwidth is critical to achieve scalable performance. The current GPUs leverage high-end memory technology (e.g., graphics DDR or GDDR) to support high compute throughput. This solution limits the memory capacity available in a GPU platform to an amount much smaller than the capacities deployed in CPU-based servers today. Furthermore, increasing memory bandwidth to match the compute has pin-count and power limitations. Instead, one should explore emerging memory technologies such as 3D-stacking [12] or cache compression [5].

Large cache: As shown in Section 4.3.3, caches provide significant benefit for throughput computing applications. An example proof of our viewpoint is that GTX280 has limited on-die memories and around 40% of our benchmarks will lose the opportunity to benefit from increasing compute flops. The size of the on-die storage should match the working set of target workloads for maximum efficiency. Some workloads have a working set that only depends on the dataset and does not change with increasing core count or thread count. For today's datasets, 8MB on-die storage is sufficient to eliminate 90% of all accesses to external memory. As the data footprint is likely to increase tomorrow, larger on-die storage is necessary for these workloads to work well. Other workloads have working set scales with the number of processing threads. For these workloads, one should consider their per thread on-die storage size requirement. For today's dataset, we found most per thread working sets can be as small as a few KB to as large as 256KB. The per thread working sets are unlikely to change in the future as they are already set to scale with increased thread count.

Gather/Scatter: 42% percent of our benchmarks can exploit SIMD better with an efficient gather/scatter support. Our simulation-based analysis projects a 3X performance benefit for **SpMV** and **RC** with idealized gather operations. Idealized gather operation can simultaneously gather all elements into SIMD register in the same amount of time to load one cache line. This may require significant hardware and be impractical to build as it may require a large number of cache ports. Therefore, this represents the upper bound of the gather/scatter hardware potential. Cheaper alternatives exist. One alternative is to use multi-banking - an approach taken by GTX280. On GTX280, its local shared memory allows 16 simultaneous accesses to 16 banks in a single cycle, as long as there are no bank conflicts. However, this data structure is explicitly managed by the programmer. Another alternative is to take advantage of cache line locality by gathering - i.e. to extract all elements required the same gather from a single load of the required cache line. This approach requires shuffle logic to reorder the data within a cache line before writing into the target register. Shuffle logic is already available in general-purpose CPUs for permutation operations within SIMD registers. Our analysis shows that many throughput computing kernels have large amounts of cache line locality. For example, **Solv** accesses on average 3.6 cache lines within each 8-wide gather request. **RC** accesses on average

5 cache lines within each 16-wide gather request. Lastly, future throughput computing processors should provide improved easy-of-programming support for gather/scatter operations.

Efficient synchronization and cache coherence: Core i7 allows instructions like increment and compare&exchange to have an atomic lock prefix. GTX280 also has support for atomic operations, but only through device memory. In both CPUs and GPUs, the current solutions are slow, and more importantly do not scale well with respect to core count and SIMD width. Therefore, it is critical to provide efficient synchronization solutions in the future.

Two types of synchronizations are common in throughput computing kernels: reductions and barriers. First, reductions should provide atomicity between multiple threads and multiple SIMD lanes. For example, **Hist** loses up to 60% of SIMD efficiency because it cannot handle inter-SIMD-lane atomicity well. We recommend hardware support for atomic vector read-modify-write operations [29], which enables conflict detection between SIMD lanes as well as atomic memory accesses across multiple threads, thus achieving 54% performance improvement on four cores with 4-wide SIMD. Second, faster barrier and coherent caches become more important as core count increases and task size gets smaller. For example, in **Solv**, the average task size is only about 1000 cycles, while a barrier takes several hundred cycles on CPUs and several micro-seconds on GPUs. We recommend hardware support for fast barriers to amortize small task size and cache coherence to guarantee memory consistency between barrier invocations. In addition, we also believe that hardware accelerated task queues will improve synchronization performance even more (68% to 109% [28]).

Fixed function units: As shown in Section 4.3.6, **Bilat** can be sped up by 2X using fast transcendental operations. Texture sampling units significantly improve the performance of **GJK**. In fact, a large class of image processing kernels (i.e., video encoding/decoding) can also exploit such fixed function units to accelerate specialized operations at very low area/power cost. Likewise, Core i7 introduced a special purpose CRC instruction to accelerate the processing of CRC computations and the upcoming 32nm version will add encryption/decryption instructions that accelerate key kernels by 10X [36]. Future CPUs and GPUs will continue this trend of adding key primitives for developers to use in accelerating the algorithms of interest.

6. RELATED WORK

Throughput computing applications have been identified as one of the most important classes of future applications [6, 10, 13, 44]. Chen et al. [13] describe a diverse set of emerging applications, called RMS (Recognition, Mining, and Synthesis) workloads, and demonstrate that its core kernel functions exist in applications across many different domains. The PARSEC benchmark discusses emerging workloads and their characteristics for CMPs [10]. The Berkeley View report illustrates 13 kernels to design and evaluate throughput computing models [6]. The UIUC's Parboil benchmark tailors to capture the strengths GPUs [44]. In this paper, we share the vision that throughput computing will significantly impact future computing paradigms, and analyze the performance of a representative subset of kernels from these workload suites on common high-performance architectures.

Multi-core processors are a major architectural trend in today's general-purpose CPUs. Various aspects of multi-core architectures such as cache/memory hierarchy [11], on-chip interconnect [4] and power management [37] have been studied. Many parallel kernels have also been ported and optimized for multi-core systems, some of which are similar to the kernels discussed in this paper [15, 17, 50]. However, these efforts concentrate on (1) overcoming parallel scalability bottlenecks, and (2) demonstrating multi-core performance over a single-core of the same type.

General-purpose computation on graphics hardware (GPGPU) has been an active topic in the graphics community. Extensive work has recently been published on GPGPU computation; this is summarized well in [3, 33]. A number of studies [8, 9, 19, 20, 21, 25, 27, 34, 40, 43, 53] discuss similar throughput computing kernels as in this paper. However, their focus is to map non-graphic applications to GPUs in terms of algorithms and programming models

Analytical models of CPUs [51] and GPUs [22] have also been proposed. They provide a structural understanding of throughput computing performance on CPUs and GPUs. However, (1) each discusses either CPUs or GPUs only, and (2) their models are very simplified. Further, they try to verify their models against real silicons, rather than to provide in-depth performance comparison between CPUs and GPUs.

This paper provides an architectural analysis of CPUs and GPUs. Instead of simply showing the performance comparison, we study how architectural features such as core complexity, cache/buffer design, and fixed function units impact throughput computing workloads. Further, we provide our recommendation on what architecture features to improve future throughput computing architectures. To the best of our knowledge, this is the first paper that evaluates CPUs and GPUs from the perspective of architecture design. In addition, this paper also presents a fair comparison between performance on CPUs and GPUs and dispels the myth that GPUs are 100x-1000x faster than CPUs for throughput computing kernels.

7. CONCLUSION

In this paper, we analyzed the performance of an important set of throughput computing kernels on Intel Core i7-960 and Nvidia GTX280. We show that CPUs and GPUs are much closer in performance (2.5X) than the previously reported orders of magnitude difference. We believe many factors contributed to the reported large gap in performance, such as which CPU and GPU are used and what optimizations are applied to the code. Optimizations for CPU that contributed to performance improvements are: multithreading, cache blocking, and reorganization of memory accesses for SIMD-ification. Optimizations for GPU that contributed to performance improvements are: minimizing global synchronization and using local shared buffers are the two key techniques to improve performance. Our analysis of the optimized code on the current CPU and GPU platforms led us to identify the key hardware architecture features for future throughput computing machines – high compute and bandwidth, large caches, gather/scatter support, efficient synchronization, and fixed functional units. We plan to perform power efficiency study on CPUs and GPUs in the future.

8. REFERENCES

[1] CUDA BLAS Library. http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/ CUBLAS_Library_2.1.pdf, 2008.

[2] CUDA CUFFT Library. http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/ CUFFT_Library_2.1.pdf, 2008.

[3] General-purpose computation on graphics hardware. http://gpgpu.org/, 2009.

[4] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, 2009.

[5] A. R. Alameldeen. *Using compression to improve chip multiprocessor performance*. PhD thesis, Madison, WI, USA, 2006. Adviser-Wood, David A.

[6] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. *Technical Report UCB/EECS-183*, 2006.

[7] D. H. Bailey. A high-performance fft algorithm for vector supercomputers-abstract. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, page 114, Philadelphia, PA, USA, 1989. Society for Industrial and Applied Mathematics.

[8] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, 2009.

[9] C. Bennemann, M. Beinker, D. Egloff, and M. Gauckler. Teraflops for games and derivatives pricing. http://quantcatalyst.com/download.php?file=DerivativesPricing.pdf.

[10] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[11] S. Biswas, D. Franklin, A. Savage, R. Dixon, T. Sherwood, and F. T. Chong. Multi-execution: multicore caching for data-similar executions. *SIGARCH Comput. Archit. News*, 37(3):164–173, 2009.

[12] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, L. Jiang, G. H. Loh, D. McCaule, P. Morrow, D. W. Nelson, D. Pantuso, P. Reed, J. Rupley, S. Shankar, J. Shen, and C. Webb. Die stacking (3d) microarchitecture. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–479, Washington, DC, USA, 2006. IEEE Computer Society.

[13] Y. K. Chen, J. Chhugani, P. Dubey, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. D. Nguyen, M. Smelyanskiy, and M. Smelyanskiy. Convergence of recognition, mining, and synthesis workloads and its implications. *Proceedings of the IEEE*, 96(5):790–807, 2008.

[14] Y.-K. Chen, J. Chhugani, C. J. Hughes, D. Kim, S. Kumar, V. W. Lee, A. Lin, A. D. Nguyen, E. Sifakis, and M. Smelyanskiy. High-performance physical simulations on next-generation architecture with many cores. *Intel Technology Journal*, 11, 2007.

[15] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core simd cpu architecture. *PVLDB*, 1(2):1313–1324, 2008.

[16] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[17] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009.

[18] M. Frigo, Steven, and G. Johnson. The design and implementation of fftw3. In *Proceedings of the IEEE*, volume 93, pages 216–231, 2005.

[19] L. Genovese. Graphic processing units: A possible answer to HPC. In *4th ABINIT Developer Workshop*, 2009.

[20] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, NY, USA, 2006. ACM.

[21] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[22] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *SIGARCH Comput. Archit. News*, 37(3):152–163, 2009.

[23] Intel Advanced Vector Extensions Programming Reference.

[24] Intel. SSE4 Programming Reference. 2007.

[25] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 185–196, Washington, DC, USA, 2005. IEEE Computer Society.

[26] J. R. Johnson, R. W. Johnson, D. Rodriquez, and R. Tolimieri. A methodology for designing, modifying, and implementing fourier transform algorithms on various architectures. *Circuits Syst. Signal Process.*, 9(4):449–500, 1990.

[27] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *ACM SIGMOD*, 2010.

[28] S. Kumar, C. J. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 162–173, New York, NY, USA, 2007. ACM.

[29] S. Kumar, D. Kim, M. Smelyanskiy, Y.-K. Chen, J. Chhugani, C. J. Hughes, C. Kim, V. W. Lee, and A. D. Nguyen. Atomic vector operations on chip multiprocessors. In *ISCA '08: Proceedings of the 35th International Symposium on Computer Architecture*, pages 441–452, Washington, DC, USA, 2008. IEEE Computer Society.

[30] N. Leischner, V. Osipov, and P. Sanders. Fermi Architecture White Paper, 2009.

[31] P. Lyman and H. R. Varian. How much information. http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/, 2003.

[32] NVIDIA. NVIDIA CUDA Zone. http://www.nvidia.com/object/cuda_home.html, 2009.

[33] Owens, D. John, Luebke, David, Govindaraju, Naga, Harris, Mark, Kruger, Jens, Lefohn, E. Aaron, Purcell, and J. Timothy. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, March 2007.

[34] V. Podlozhnyuk and M. Harris. Monte Carlo Option Pricing. http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MonteCarlo/doc/MonteCarlo.pdf.

[35] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[36] R. Ramanathan. Extending the world.s most popular processor architecture. *Intel Whitepaper*.

[37] K. K. Rangan, G.-Y. Wei, and D. Brooks. Thread motion: fine-grained power management for multi-core systems. *SIGARCH Comput. Archit. News*, 37(3):302–313, 2009.

[38] R. Sathe and A. Lake. Rigid body collision detection on the gpu. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters*, page 49, New York, NY, USA, 2006. ACM.

[39] N. Satish, M. Harris, and M. Garland. Designing efficient sorting algorithms for manycore GPUs. In *IPDPS*, pages 1–10, 2009.

[40] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey. Fast Sort on CPUs and GPUs: A Case For Bandwidth Oblivious SIMD Sort. In *ACM SIGMOD*, 2010.

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

[42] M. Silberstein, A. Schuster, D. Geiger, A. Patney, and J. D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In *Proceedings of the 22nd ACM International Conference on Supercomputing*, pages 309–318, June 2008.

[43] M. Smelyanskiy, D. Holmes, J. Chhugani, A. Larson, D. Carmean, D. Hanson, P. Dubey, K. Augustine, D. Kim, A. Kyker, V. W. Lee, A. D. Nguyen, L. Seiler, and R. A. Robb. Mapping high-fidelity volume rendering for medical imaging to cpu, gpu and many-core architectures. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1563–1570, 2009.

[44] The IMPACT Research Group, UIUC. Parboil benchmark suite. http://impact.crhc.illinois.edu/parboil.php.

[45] J. Tolke and M. Krafczyk. TeraFLOP computing on a desktop pc with GPUs for 3D CFD. In *International Journal of Computational Fluid Dynamics*, volume 22, pages 443–456, 2008.

[46] N. Univ. of Illinois. Technical reference: Base operating system and extensions , volume 2, 2009.

[47] F. Vazquez, E. M. Garzon, J.A.Martinez, and J.J.Fernandez. The sparse matrix vector product on GPUs. Technical report, University of Almeria, June 2009.

[48] V. Volkov and J. Demmel. LU, QR and Cholesky Factorizations using Vector Capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[49] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[50] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

[51] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[52] W. Xu and K. Mueller. A performance-driven study of regularization methods for gpu-accelerated iterative ct. In *Workshop on High Performance Image Reconstruction (HPIR)*, 2009.

[53] Z. Yang, Y. Zhu, and Y. Pu. Parallel Image Processing Based on CUDA. In *International Conference on Computer Science and Software Engineering*, volume 3, pages 198–201, 2008.