RC25033 (W1008-020) August 5, 2010
Computer Science

IBM Research Report

Can CPUs Match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU

Rajesh Bordawekar, Uday Bondhugula, Ravi Rao
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Can CPUs match GPUs on Performance with Productivity?: Experiences with Optimizing a FLOP-intensive Application on CPUs and GPU

Rajesh Bordawekar Uday Bondhugula Ravi Rao
IBM T. J. Watson Research Center
bordaw,ubondhug,ravirao@us.ibm.com

Abstract

In this work, we evaluate performance of a real-world image processing application that uses a cross-correlation algorithm to compare a given image with a reference one. The algorithm processes individual images represented as 2-dimensional matrices of single-precision floating-point values using $\Theta(n^4)$ operations involving dot-products and additions. We implement this algorithm on a NVIDIA Fermi GPU (Tesla 2050) using CUDA, and also manually parallelize it for the Intel Xeon X5680 (Westmere) and IBM Power7 multi-core processors. Pthreads and OpenMP with SSE and VSX vector intrinsics are used for the manually parallelized version on the multi-core CPUs. A number of optimizations were performed for the GPU implementation on the Fermi, including blocking for Fermi's configurable on-chip memory architecture. Experimental results illustrate that on a single multi-core processor, the manually parallelized versions of the correlation application perform only a small order of factor slower than the CUDA version executing on the Fermi – 1.005s on Power7, 3.49s on Intel X5680, and 465ms on Fermi. On a two-processor Power7 system, performance approaches that of the Fermi (650ms), while the Intel version runs in 1.78s. These results conclusively demonstrate that performance of the GPU memory subsystem is critical to effectively harness its computational capabilities. For the correlation application, a significantly higher amount of effort was put into developing the GPU version when compared to the CPU ones (several days against few hours). Our experience presents compelling evidence that performance comparable to that of GPUs can be achieved with much greater productivity on modern multi- core CPUs.

1. Introduction

One of the game-changing technical events in the recent past has been the emergence of Graphics Processing Units (GPUs) as a viable platform for general purpose scientific computing. With the transistor count doubling every six months and a reasonable amount of on-card memory (currently from 1 GB to 6 GB), GPU-based cards have brought TFLOPS-scale performance computing to the masses. Availability of cheaper and faster GPUs from NVIDIA and AMD, along with associated programming models such as CUDA [4] and OpenCL [2] have further led to increased usage of GPU-based systems.

In the mean time, there has also been tremendous progress in conventional CPU processor architectures. The current multi-core processors exhibit multiple (at least four) powerful cores, deeper (currently 3-level), bigger, and faster caches, higher clock frequency, and higher on-chip memory bandwidth. These processors also possess improved hardware support for floating point operations and parallel execution, e.g., short-vector data parallelism, and simultaneous multi-threading. Recent advances in compilation technology, especially in auto-parallelization and simdization, have also enabled efficient utilization of these resources for applications in the scientific domain. Thus, the modern CPU, with its hardware and software infrastructure, has emerged as a formidable competitor to the GPU, even for traditional scientific applications.

Given wide-spread availability of systems built from commodity multi-core processors along with inexpensive and powerful GPUs, a key question to be addressed is whether it is *worth* parallelizing a FLOP-intensive scientific application on a multi-core CPU or accelerate it on a GPU (the latest NVIDIA GTX480 is currently available for around $500 for example). This paper tries to answer this question by describing our experiences in parallelizing a real-world image processing application from the computational biology domain. This application performs cross-correlation operations on 2-dimensional images to compute spatial correlation statistics of natural scenes. The core kernel involves $\Theta(n^4)$ computation over 2-dimensional single-precision floating point matrices, where each step involves 4 multiplications and 4 additions. We implemented the GPU version of the algorithm using CUDA 3.0 and ran it on a NVIDIA Fermi (Tesla 2050). Our Fermi implementation uses a massively-parallel tiled algorithm that organizes the computations among threads so as to exploit different types of memory systems. We also developed multi-threaded vectorized versions of the code using pthreads and OMP, with VSX/SSE, and ran them on the Intel Xeon X5680 (Westmere) and IBM Power7 processors.

Our experimental results illustrate that the best performing GPU version (465ms) fares better than the multi-threaded versions executing on a single processor system: the Power7 version required 1.005s (32 OpenMP threads) and the Intel version took 3.49s (12 OpenMP threads). When two processors were used, the Power7 version approached the Fermi performance (650ms using 64 OpenMP threads), while the Intel version ran for 1.78s (24 OpenMP threads). While Fermi's new features, specifically, the configurable memory sub-system, led to better utilization of its computing resources, it has also increased the complexity of the CUDA code development and optimizations. While it took us several weeks to optimize the GPU code, it took a few hours to develop the optimized versions for multi-core CPUs. Our experience has shown that one can get comparable performance on multi-core CPUs with considerable less work. Although these results are from a single application, they expose a number of traits and fundamental performance issues while optimizing FLOP-intensive high-reuse applications on GPUs.

This paper makes the following contributions:

- First, we describe in detail various steps in developing an optimized CUDA application on the Fermi GPU and present a

tiled version of the correlation algorithm that uses parameterized two-dimensional iteration space blocking.

- Second, we discuss various Fermi-specific memory optimizations, e.g., choosing a suitable memory layout for data structures and identifying the optimal memory configuration.
- Finally, we perform extensive experimental evaluation of the CUDA version and compare it against multi-threaded SIMDized versions of the correlation algorithm on two different multi-core architectures: IBM Power7, and Intel Xeon (Westmere).

The rest of the paper is organized as follows. Section 2 discusses the correlation application. Section 3 describes its implementation using CUDA on the NVIDIA Fermi GPU. Section 4 presents the experimental performance results of both the GPU and CPU implementations. We conclude in Section 5.

2. Description of the Application

In this study, we investigate parallelization of a computational biology application that computes spatial correlations for large image datasets derived from natural scenes [6]. The motivation for performing this computation is to relate the spatial correlation statistics of natural scenes with the properties of neurons in the primate visual cortex, such as their orientation and color selectivity, and their receptive field sizes. Though the computation can be carried out relatively quickly for one dimensional scalar values such as image intensity, the computation becomes challenging when the input consists of multi-dimensional matrices over large (in the hundreds) image ensembles.

There are many techniques to compute statistical properties of interest. We use a method that involves computing an orientation or color vector at a given image location, and its correlation with other orientation or color vectors at different displacements. Here, orientation and color are represented using 2-element tuples. The orientation consists of a magnitude and direction, and can be represented in Cartesian form as (x_1, x_2). The color consists of a triplet in $L(a, b)$ color space, where L is the luminance. We use a 2-dimensional matrix consisting of the 2-element tuples (a, b), which represents the color in the $a - b$ plane. There are many techniques to compute orientation and color, and there are multiple parameters involved, such as the filter sizes used to estimate orientation [7]. We would like to compute the joint orientation and color distribution repeatedly over a large image ensemble while varying the different methods, and their parameters. At present, the sequential algorithm processes images from an ensemble of 100+ images, one at a time. For each image, the correlation algorithm computes a small correlation matrix. These correlation matrices are then further analyzed to explore statistical properties of the natural scenes. As we shall see, the core correlation computation is extremely expensive and any improvement in its performance, would significantly improve overall running time of processing an ensemble of images.

Core computation kernel The cross-correlation kernel processes an 500×500 pixel image using four input data sets, representing 2-element color and orientation tuples of each image pixel. Each data set represents a two-dimensional space of size 500×500 using single-precision floating point numbers. Thus, total space consumption per image is 4 MB. The result of the correlation is a 250×250 single-precision matrix, consuming 250 KB of space.

The cross-correlation algorithm compares the base image against its shifted version, both in X- and Y-dimensions (Figure 1(a)). In both dimensions, the extent of the shift is 250 (henceforth, referred to as the CORR_SIZE), leading to a two-dimensional correlation space of 250×250 (62500) positions. For each position in the correlation space, the overlapped sections of the base image, called the *src*, and its shifted version, called the *mask*, are correlated (shaded region in Figure 1(b)). Figure 2 presents the corresponding pseudo-code. The correlation computation is implemented using 4-level nested loops over 4 arrays: `color1`, `color2`, `orientation1`, and `orientation2`. The outermost two loops correspond to shifts in the two-dimensional correlation space, while the innermost two loops correspond to computation over individual data sets. In every step, two elements of each array are fetched with two different offsets: `src_index` and `mask_index`. The correlation operation implements a pair-wise dot-product over aligned pair of values from src and mask from overlapped sections of every data set and aggregates the computed dot-product values to determine the final correlated value for that shift (Figure 2, lines 11–15).

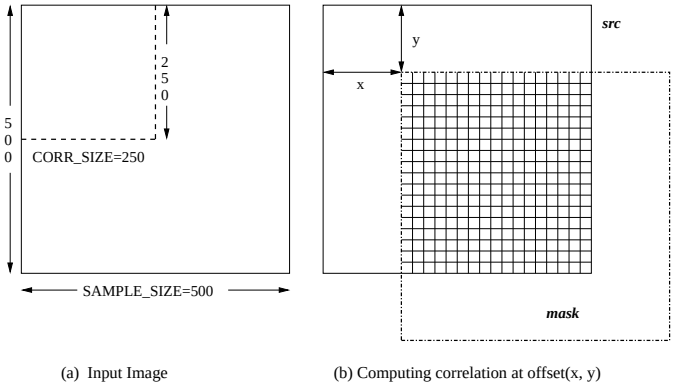


Figure 1. Computing 2-dimensional correlation

Thus, the correlation algorithm executes in a (250×250) *iteration* space, where each iteration operates over a (500×500) data space. The amount of work per iterations varies according to its corresponding offsets. Hence, the overall computation requires $\Theta(n^4)$ steps, where n is 500. Each step requires 4 single-precision floating point multiplications and 3 single-precision floating point additions. In addition, there are $(250 \times 250 \times \Theta(n^2))$ additions (when parallelized, 250×250 collective reduction operations, each adding $\Theta(n^2)$ dot-products) for computing the final result.

The rest of the paper focuses on evaluating different strategies for parallelizing this correlation algorithm on the NVIDIA Fermi GPU and multi-core CPUs.

3. Implementing the Correlation Algorithm on Fermi

This section describes various steps in porting the correlation algorithm on the NVIDIA Fermi architecture using CUDA.

3.1 Determining the Thread Mapping

The first step in developing any CUDA application is to determine the number and mapping of the CUDA threads used in the computation. The correlation algorithm (Figure 2) exhibits two key characteristics: (1) the computation is inherently unbalanced; the amount of computation required for the correlation offset (0,0) is four times that for the correlation offset (249, 249), and (2) the iteration space, (250, 250), is smaller than the data space, (500, 500). Therefore, if we were to use 500×500 threads, with one thread per element in the data space, many threads will have no work due to the unbalanced nature of the computation. Also, computing every result value requires coordination among all participating threads, leading to excessive synchronization costs.

Therefore, we decided to use 250×250 threads to parallelize the correlation computation (Figure 3: A). Each thread would be busy throughout the application, but each thread would perform a

```
1 for(offset_y=0; offset_y < CORR_SIZE; offset_y++) {
2     for(offset_x=0; offset_x < CORR_SIZE; offset_x++) {
3         src_index = 0;
4         mask_index = 0;
5         dot_product = 0.0f;
6         for(rows=0; rows < SAMPLE_SIZE−offset_y; rows++) {
7             for(columns=0; columns < SAMPLE_SIZE−offset_x; columns++) {
8                 src_index = (offset_y + rows)*SAMPLE_SIZE + offset_x + columns;
9                 mask_index = rows*SAMPLE_SIZE + columns;
10                correlation_index = offset_y*CORR_SIZE + offset_x;
11                dot_product = (color1[src_index]*color1[mask_index])+
12                (color2[src_index]*color2[mask_index])+
13                orientation1[src_index]*orientation1[mask_index])+
14                orientation2[src_index]*orientation2[mask_index]);
15                correlation_array[correlation_index]+=dot_product;
16            }
17        }
18    }
19 }
```

Figure 2. Pseudo-code for the two-dimensional correlation algorithm

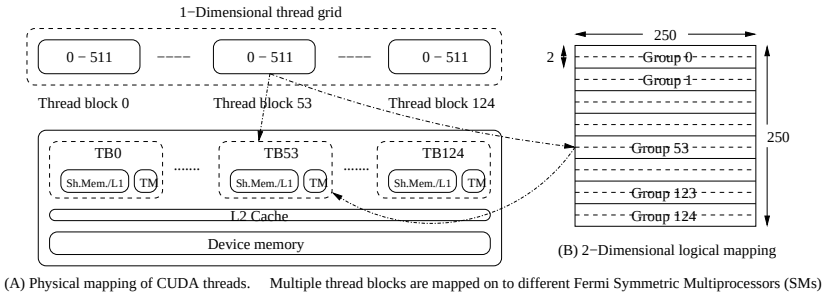


Figure 3. Thread mapping for the CUDA implementation

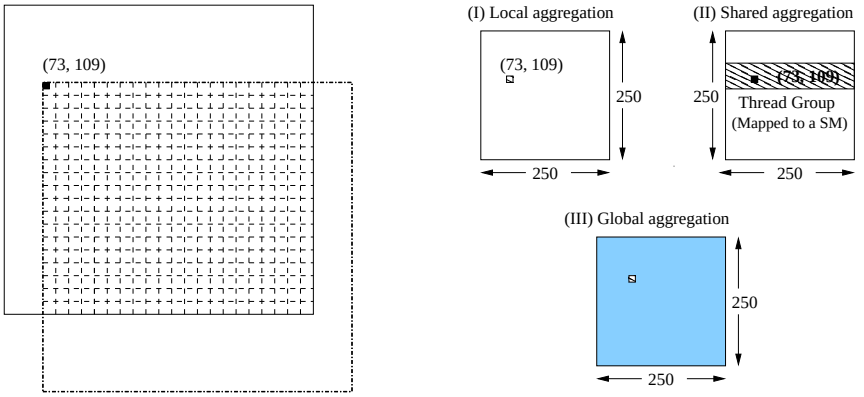
different amount of work. The 250×250 threads are allocated as a 1-dimensional grid of 125 thread blocks, each with 512 threads(each SM can hold upto 512 threads). Each thread block is assigned to a separate symmetric multiprocessor (SM). Logically, the 250×250 threads are viewed as a 2-dimensional grid and assigned an ID in the $(250, 250)$ space (Figure 3:B). Thus, each thread maps logically to a unique point in the iteration space.

3.2 Selecting Aggregation Strategy

Given the thread mapping, the correlation algorithm can be implemented in three different ways, depending on how aggregation is performed: (1) local aggregation, (2) shared aggregation, and (3) global aggregation. These three approaches primarily differ in the way intermediate results are added to compute the correlation value at any offset (x, y). Figure 4 illustrates computation of correlation at offset $(73, 109)$ using the three different implementations.

In the shared aggregation approach, a logical thread group (Figure 3(b)) is assigned the corresponding set of offsets in the 2-dimensional correlation space. Moreover, computation of each offset in the assigned set is executed in parallel by all 500 threads in the thread group. For an offset in the correlation space, all 500 threads from the corresponding thread group read the relevant portions of the four arrays in parallel, perform the dot-products and additions, and then atomically update (via the CUDA `atomicAdd()` function) a variable stored in the shared memory to maintain the running sum (Figure 2: lines 11–14). Once all threads have completed their work, one thread in the thread group stores the final sum to the corresponding offset in the correlation array in device memory without any synchronization (Figure 2: line 15). Finally, in the global aggregation approach, computation of each offset in the correlation space is implemented using all 250×250 threads. For each correlation offset, every thread in the application fetches relevant portions of the arrays in parallel, performs dot-products and additions, and then uses `atomicAdd()` to update a shared variable in the shared memory of the corresponding symmetric multiprocessor. Once all threads within a thread block have completed their work, one thread updates the correlation array in the device memory using the `atomicAdd()` function. While this approach enables maximum parallelism with respect to memory accesses and computation, it suffers from maximum synchronization overhead as it needs to update shared variables both in shared and device memories.

As both the shared and global aggregation approaches require synchronization, we use the local aggregation approach in our implementation. In the local aggregation approach, a thread at position (x, y) in the $(250, 250)$ logical map is responsible for computing the correlation at offset (x, y) in the $(250, 250)$ correlation space. Each thread serially fetches data required for computing correlation at that offset. For example, for computing correlation at offset (73, 109), a thread with logical ID (73,109) fetches all elements in the shared region of the four arrays. It performs the necessary dot-products and additions (Figure 2: lines 11-14) and updates a



(a) Computing correlation at offset (73, 109)

(b) Three implementation alternatives

Figure 4. Three alternative CUDA implementations. The shaded regions in (b) represent the threads involved in computing the correlation at the offset (73,109).

register variable, `dot_product` to maintain a running sum. Once the computation is completed, the thread updates the corresponding position in the correlation array in the device memory (Figure 2: line 15) *without any synchronization*. Thus, this approach is an embarrassingly parallel implementation of the correlation algorithm in which the iteration space defined by the outer two loops of the correlation kernel (Figure 2) get distributed over $\Theta(n^2)$ threads. All threads then compute results for their corresponding offsets using $\Theta(n^2)$ concurrent computations, without the need for any coordination on global or shared memory variables.

3.3 Blocking for On-chip Memories

The next step is to optimize memory accesses for the local aggregation approach. The Fermi architecture, in addition to the traditional GPU memories (e.g., device and texture memories), also supports a configurable memory architecture [5], in which a 64 KB memory region can be configured either as 48 KB shared memory or 16 KB L1 cache, or vice versa (Figure 3). We can use the higher shared memory capacity to exploit data reuse that is abundant in the correlation algorithm. The correlation algorithm reads each image using four data sets, each representing a 500×500 single precision float array. Thus, the total memory usage per image is 4 MB. The result is stored in a 250×250 single-precision float array which consumes 250 KB of space. Neither the input arrays nor the result array can be stored in entirety in Fermi's shared memory. However, analysis of the correlation kernel (Figure 2) reveals several opportunities for data reuse across threads when parallelized using the local aggregation method (Figure 5).

Figure 5(1) presents a thread group with 500 threads, logically mapped as a (250,2) region in the (250, 250) iteration space. Let i and $i+1$ be the row indices of the two 250 thread rows. Let us consider four threads, two in each row, with positions (i, j), $(i, j+2)$, $(i+1, j)$, and $(i+i, j+2)$. The thread (i, j) computes the correlation using two equal-sized regions, $(500 - i) \times (500 - j)$, of the *src* and *mask* arrays (Figure 5(2) and (3)). However, these two regions differ in the starting offset: the *src* region starts at offset (i, j), and the *mask* region starts at offset $(0, 0)$. The thread $(i + 1, j)$ uses two smaller-sized regions of size, $(500 - i - 1) \times (500 - j)$; the *src* region starts at offset $(i + 1, j)$, but the *mask* region starts at offset $(0, 0)$. Thus, both threads use the same first j elements of the 0^{th} row of the *mask* arrays for processing the 0^{th} row of their *src* regions (note that, the 0^{th} row of the two *src* regions differ). In general, a k^{th} *mask* row is accessed by all threads while processing the k^{th} row of their *src* regions. Figure 5(A) illustrates this *mask-row reuse*. In addition, multiple elements in a *mask* or *src* row are reused by threads that lie in the same row of a thread group. For example, the two darkened regions in the first row of mark and *src* arrays are shared by threads at positions (i, j) and $(i, j + 2)$ (Figure 5(B)). This is an example of the *intra-row reuse*. Finally, as shown in Figure 5(2), the *src* region of thread (i, j) overlaps with the *src* region of thread $(i + 1, j)$. Thus, the row $i + 1$ fetched for thread $(i+1, j)$ could be used by thread (i, j) for its next iteration. In fact, all *src* rows fetched by the thread $(i+1, j)$ could be reused by the thread (i, j), and also by other threads with the same row offset i. This is an example of the *inter-iteration reuse*.

To exploit these data reuse opportunities on the Fermi, we have developed a new implementation that uses two-dimensional tiling to exploit the limited amount of shared memory. In our implementation, the outermost two loops of the correlation kernel (Figure 2) are tiled (or blocked) such that the (250*250) iteration space is partitioned into groups of iterations arranged as rectangular tiles of volume 500. Thus, each tile matches a thread group that gets assigned to a separate symmetric multiprocessor (Figure 3:B) and can be executed concurrently. Also, by utilizing 500 threads, we are able to exploit maximal parallelism while reading the input data. Concurrent accesses by logically contiguous threads get optimized via hardware coalescing.

The iteration space spanned by a tile gets mapped to the (250*250) iteration space in a particular layout order (e.g., left-to-right, top-to-bottom). While the tile size is fixed, the tile shape can be varied to exploit different types of data reuse. Figure 6 presents some of the tile shapes explored by our implementation. The tile shape and its position in the (250*250) iteration space determine the amount and type of data to be accessed. By increasing the height, i.e., the number of rows of a tile, the amount of mask-row reuse is increased, as larger number of threads with different row indices can reuse the same mark row for their computations. Figure 6 presents the amount of mask-row reuse per row for different tile shapes. Increasing the tile height also increases the amount of inter-iteration reuse with the *src* arrays. However, as the tile size is fixed, by increasing the tile height, the tile width reduces in proportion. This, in turn, reduces the amount of intra-row reuse in the *mask* and *src* arrays. While the *slimmer* tiles (with more rows than columns) increase data reuse, they also incur additional memory access costs as threads now generate strided accesses to non-contiguous memory regions.

Figure 7 presents the pseudo-code of the two-dimensional tiled implementation of the local aggregation approach. This kernel rep-

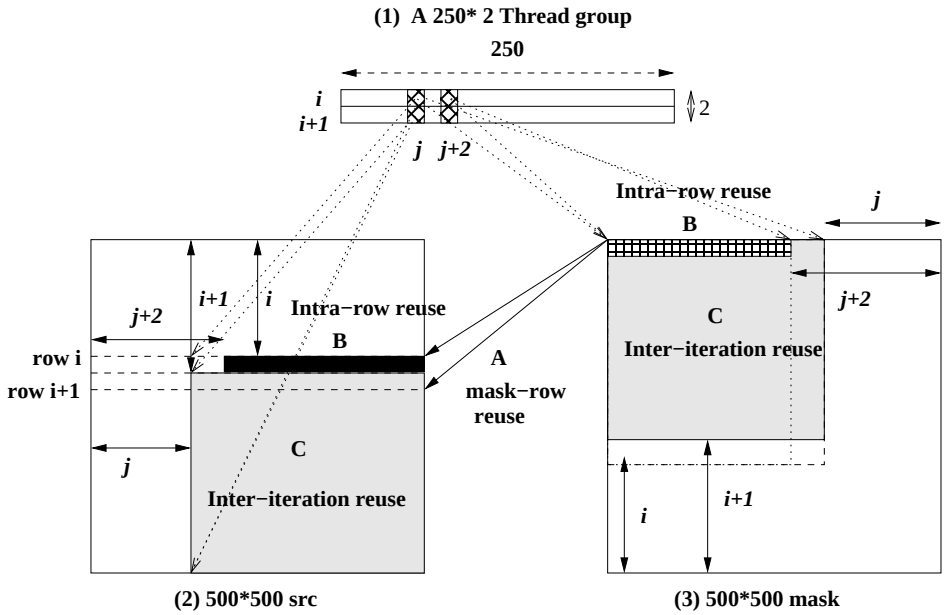


Figure 5. Data reuse in the parallelized correlation kernel: (A) Mask-row reuse, (B) Intra-row reuse, and (C) Inter-iteration reuse

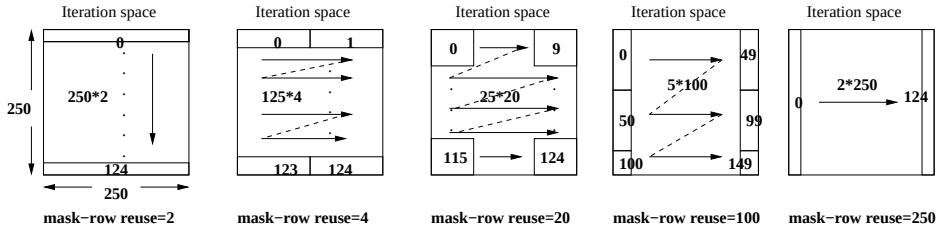


Figure 6. Two-dimensional blocked mapping onto a threadblock

```
1 Input: tile_x, tile_y (# tiles in x and y dimensions)
2 Input: block_x, block_y (tile size in x and y dimension)
3 Compute thread ID (tid) and thread-block ID (blockID)
4 threadGrp_x = blockID % tile_x, threadGrp_y = blockID / tile_x
5 myXOff = tid % block_x+ threadGrp_x*block_x;
6 myYOff = tid / block_x+ threadGrp_y*block_y;
7 corr_offset = threadGrp_y*block_y*CORR_SIZE+(tid/block_x)*CORR_SIZE+myXOff;
8 myYCount = SAMPLE_SIZE-myYOff;
9 dotproduct = 0;
10 for(j=0; j < SAMPLE_SIZE-(threadGrp_y*block_y); j++) {
11     Fetch the j^th row of the mask array into shared memory
12     Fetch block_y rows of the src array into shared memory (FOR EXCLUSIVE SHARED MEMORY VERSIONS ONLY)
13     __syncthreads();
14     if (((myXOff < 250) && (myYOff < 250)) && (j < myYCount)){
15         for(i=0; i < SAMPLE_SIZE-myXOff; i++){
16             Compute the src_offset using i and j
17             Fetch src data from texture memory or caches at the src_offset (or shared memory)
18             Compute correlation using the src and mask elements (lines 11--14, Figure 2)
19             dotproduct += computed correlation
20         }
21     }
22 }
23 corr[corr_offset]=dotproduct
```

Figure 7. Pseudo-code for the 2-dimensional tiled CUDA version of the correlation algorithm using the local aggregation approach

resents the $\Theta(n^2)$ computation performed by each thread concurrently. The kernel takes the tile shape and number of tiles as input parameters. On line 8, each thread computes the offset in the result `corr` array where the final correlation value needs to be stored. On line 9, each thread calculates the required number of iterations on the y-axis which translates to the number of rows of the *src* array to be fetched. Lines 11–21 illustrate the implementation of the $\Theta(n^2)$ computation. All threads in a tile execute the outermost loop; the number of iterations is dependent on the logical position of that tile on the y-axis. Each iteration of the outermost loop fetches a row of the *mask* array, beginning at offset 0, and stores it in the shared memory (line 12). In the innermost kernel, each thread first computes the offset of the corresponding *src* data and fetches data either using the texture memory or caches. Using the *mask* and *src* data, it computes the correlation and updates the running sum, `dotproduct` (line 19). The `dotproduct` value is finally stored in the resultant `corr` array. While all 500 threads in a tile execute the outermost loop, and fetch the *mask* row, number of invocations of the innermost loop vary as per the thread's position (line 14). Note that *src* data can be alternatively fetched into the shared memory and used later. Our implementation exploits the inter-iteration reuse to minimize shared memory accesses. Only the first iteration fetches `block_y` rows; all future iterations fetch only one row and reuse the previously fetched `block_y-1` rows.

3.4 Optimizing Input Data Layout

The final step involves optimizing accesses to the *src* and *mask* arrays. Recall that the application uses four 500×500 single-precision float arrays as input. In our implementation, we use the same read-only data structures for both the *src* and *mask* arrays. While our proposed thread-mapping can enable cross-thread hardware coalescing while accessing a single array, strided accesses to multiple arrays cannot be optimized using hardware coalescing. Accesses to individual arrays can be further improved by *co-locating* (packing) four single-precision float arrays into a single large `float4` array: the i^{th} `float4` element of the co-located array consists of i^{th} elements of the four input arrays. Thus, a single memory access to a co-located array can return a `float4` element that contains four single-precision float elements from four different arrays. Since the input data arrays are read-only, they can be also accessed using GPU's texture memory. On Fermi, either the texture memory or L1/L2 caches can also be used for optimizing strided memory accesses. Thus, for the correlation algorithm, we have many different memory access alternatives based on how input data is stored, i.e., separate and co-located arrays, each accessed using either the L1/L2 caches, texture memory, shared memory, or device memory.

4. Experimental Evaluation

All parallel GPU and CPU versions of the correlation application share the same sequential pre-processing component. This component reads the image data from four input files (two each for color and orientation components), and populates the four 500×500 single-precision floating point arrays. Once the arrays are populated, the algorithm performs a pass over the image data in which color and orientation data sets are normalized. The normalized image data is then processed by the correlation kernel either using a single packed array or four separate data arrays.

Datasets The original correlation algorithm processes images from an ensemble of 100+ images. The standard size of all images in the datasets is 500×500 pixels [6], and each image is represented by four files of 1 MB each. We ran our tests on twelve randomly selected images from the repository. The images share many similarities in the color and orientation features, hence individual execution performance on every image is very similar. Hence, results are presented for only one image.

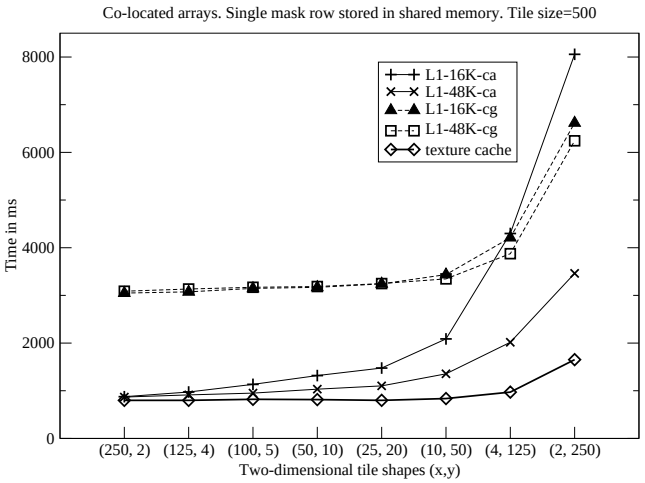


Figure 8. Performance of the tiled correlation algorithm on co-located arrays. A single *mask* row is stored in the shared memory

4.1 Evaluation on NVIDIA Fermi

We have implemented the local aggregation version of the correlation algorithm using CUDA 3.0 and evaluated it on a Linux server hosting a NVIDIA Tesla 2050 GPU.

The NVIDIA Tesla 2050 GPU is based on the new generation Fermi architecture [5]. The Fermi architecture incorporates several architectural improvements over the previous generations of NVIDIA GPUs (e.g., the GTX 200 series). The key improvements include: larger number of cores per SM (32), improved double precision performance, faster atomic operations, support for single- and double-precision FMA instructions, concurrent kernel execution, and improved ECC support [5]. However, the most important feature of the Fermi architecture is the configurable memory subsystem with true cache hierarchy and larger shared memory. The Fermi architecture provides a 64 kB configurable memory region within each SM, that can be split as a 16 KB cache or 48 KB shared memory or vice versa. In addition to the per-SM local caches, there is a 768 KB unified cache that is shared across SMs and is also used as a second-level texture memory cache 3. The CUDA 3.0 programming model enables the user to modify the memory subsystem for a specific kernel. In addition, the CUDA 3.0 compiler provides a flag (`-dlcm`) to control how the global memory accesses are cached. The (`-dlcm=ca`) configuration directs the CUDA runtime to cache the global memory accesses in both L1 and L2 caches, whereas the (`-dlcm=cg`) configuration directs the CUDA runtime to use the L2 caches only.

In our CUDA implementation, the correlation kernel fetches the four input data sets either as four separate arrays or as a single co-located array, and returns a single result array. In both cases, we have observed that the cost of data transfer is minuscule (0.05% of the overall kernel execution time, as reported by the CUDA Performance Analyzer). A majority of the time spent is spent in the core computational kernel of the CUDA correlation algorithm (Figure 2).

In our first experiment, we evaluate the efficacy of two-dimensional tiling. Recall that the two-dimensional tiling was implemented to exploit the reuse of a *mask* row across multiple rows of the *src* array. In this version of the tiled implementation, each row of the *mask* array is brought into the shared memory (in practice, 4 rows of 4 separate arrays or a row of the co-located array) and correlated with

	Processor architectures		
	NVIDIA Fermi (Tesla 2050)	Intel Xeon E5680 (Westmere)	IBM Power7
Cores	448 (14*32)	6 cores with 2-way SMT	8 cores with 4-way SMT
Frequency	1.15 GHz	3.33 GHz	3.55 GHz
Memory	16/48 KB Shared Memory per SM 16/48 KB L1 cache per SM, 768 KB Unified L2 3 GB Device Memory	32 KB L1 I and D caches 256 KB L2 cache per core 12 MB Shared L3 cache	32 KB L1 I and D caches 256 KB L2 cache per core 32 MB L3 cache (eDRAM)
Memory bandwidth	144 GB/s	6.4 GT/s (QPI)	100 GB/s (sustained)
SIMD	Dual-issue 32-way SIMT	2 64-bit MADDs or 4 32-bit MADDs	4 32/64-bit FMAs

Table 1. Architectural summary of the three multi-core processors being used in our experiments

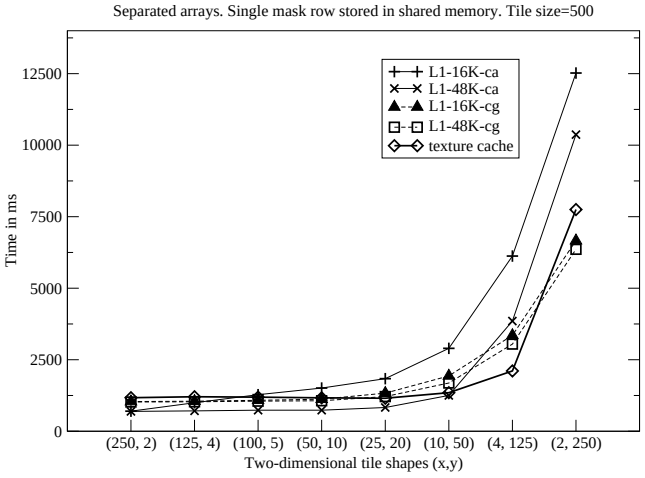


Figure 9. Performance of the tiled correlation algorithm on separated arrays. A single *mask* row is stored in the shared memory

the rows of the *src* array as defined by the iteration tile. We fetched the *src* data using either the cache hierarchy or the texture memory. Furthermore, we ran the tests with 16 and 48 KB caches, with different cache allocation compilation options (i.e., `ca` and `cg`). We used the two array layout strategies (i.e., co-located and shared), and evaluated 8 different tile shapes. Figures 8 and 9 present the performance results. From these results, it is obvious that the best performance is observed for wider tiles, e.g, for the tiles of shape, (250,2), and (125, 4). As the tiles become thinner, the amount of mask-row reuse increases as the number of rows increases. However, thinner tiles lead to threads issuing larger number of small memory accesses for non-contiguous memory regions. This increased memory access cost affects the overall performance. For both the co-located and separate arrays, the worst performance is observed for the thinnest tile, (2,250), that contains 250 2-element rows.

The results also illustrate the effect of memory layout and memory sub-system configuration on the overall performance. The cache and texture memory react differently to the two different memory layouts. Performance of cached memory versions is better when separate arrays are used. In contrast, texture memory performs better when a co-located array is used. This may be due to the different replacement policies used in the cache and texture memory hierarchies. Even when the cache memory is used, the performance behavior is different for different memory layouts. For the separate arrays scenario, in most cases, the cache size has insignificant impact on the performance. For thinner tiles, the performance of the cached implementation degrades drastically; the worst performing cases include cases compiled with the `ca` flag, which forces the runtime to use only the L1 cache. When the co-located array is used, versions that use the cache when compiled with the `cg` flag perform much slower than those compiled with the `ca` flag. This behavior is the direct result of packing the related data into one single array, thus increasing spatial locality. In addition, versions compiled with `cg` provide similar performance across a wide variety of tile sizes. This behavior again demonstrates the effects of spatial locality as large contiguous portions of the array can now be stored in the 768 KB L2 cache. When the co-located array is used, the best performance is observed when texture memory is used. This is a consequence of the texture memory being specifically designed for accelerating accesses to complex types like `float4`. For both memory layouts, the worst performance is observed when the L1 cache was 16 KB, irrespective of the compilation flag setting. Surprisingly, the texture memory provided comparable performance to the cache memory. Overall, the best performance was observed when the tile shape was wider, i.e., with fewer rows, as it resulted in fewer accesses to non-contiguous datasets. When the caches were used, the best performance (694.24ms) was observed for the (250, 2) tile, when the arrays were accessed separately, the L1 cache configured to be 48 KB, and the code compiled with `ca` flag. When texture memory was used, the best performance, 799.06ms, was again observed for the (250, 2) tile, when the input arrays were accessed as a single co-located array.

Therefore, we decided to focus on the most promising tile shapes. The next optimization is to use on-chip shared memory for accessing both the *mask* and *src* rows. Since the maximum size of the shared memory is 48 KB, it can hold at most 5 complete rows of either *mask* or *src* arrays (note that each row requires (500*4*4)=8 KB of space, and in practice, shared memory cannot hold 48 KB of data). Hence, we focus on the two tile shapes, (250, 2) and (125, 4), for the shared memory optimizations. Table 2 shows the performance of the correlation application for two tile shapes, (250, 2), and (125, 4), when both the *mask* and *src* rows were kept in the shared memory. For the tile with shape (250, 2), at a given time, 2 rows of the *src* array, and a row of the *mask* array are stored in the shared memory, while for the tile with shape (125, 4), at a given time, 4 rows of the *src* array, and a row of the *mask* array are stored in the shared memory (Figure 7, line 12). As Table 2 illustrates, for both co-located and separate array cases, the (250, 2)-shaped tile performs better, and for both tiles, performance with separate arrays is worse than when a single co-located array is used. The (125,4)-shaped tile causes more non-contiguous memory accesses than the (250, 2)-shaped tile. Furthermore, when separate arrays are used, the GPU kernel makes four different accesses to arrays stored at different memory locations. These two issues lead to degradation in overall performance. The best performance for the correlation application (465 ms), was observed when the tile with shape (250, 2) was used with shared memory to store *src* and *mask* rows from a single co-located array.

To further evaluate the performance of the (250,2) tile, we ran the kernel using only the texture memory or L1/L2 caches for storing the *mask* and *src* rows. As Table 3 illustrates, the performance degrades substantially when only texture memory or L1/L2 caches

Co-located array (time in ms)					Separate arrays (time in ms)				
Texture	Cache				Texture	Cache			
	16 KB		48 KB			16 KB		48 KB	
	ca	cg	ca	cg		ca	cg	ca	cg
1310.66	1283.79	3222.26	890.60	3339.13	1126.94	1779.36	1450.78	944.96	1399.75

Table 3. Performance of the tiled algorithm for the (250*2) tile, with both *src* and *mask* data stored in the texture or caches

	Co-located array		Separate arrays	
Tile Shape	(250, 2)	(125, 4)	(250, 2)	(125, 4)
Time (ms)	465.74	510.22	518.48	543.03

Table 2. Performance of the tiled algorithm for different tile shapes with both *src* and *mask* data stored in shared memory (48 KB) (time in ms)

were used. This result confirms that to obtain the best performance, one needs to use shared memory exclusively.

As the final experiment, we manually unrolled the computation for the tile shape (250, 2) when shared memory is used for storing both *mask* and *src* rows. We implemented two versions of unrolling: (1) The first version used 62 thread blocks, each with 512 threads. In this version the work per thread was double that of our original implementation (which used 125 thread blocks of 512 threads), and (2) The second version used 125 thread blocks with 250 threads each. Each thread block was viewed as a tile of shape (250, 1). This version also causes each thread to compute correlations for 2 offsets, but with lesser register pressure than the first unrolled implementation. The first version required 624.63 ms and the second version took 862.24 ms. The performance degradation in the first version was mainly due to register spilling that caused extra accesses to device memory. As the second version used fewer threads than our original implementation, it required extra main memory accesses to move data from device to shared memory, and with fewer thread per SM, the device memory access costs degraded the overall performance.

4.2 Multi-core CPU Implementations

In this section, we describe different strategies used to manually parallelize and optimize the original sequential program on Intel Xeon X5680 and IBM Power7 processors.

4.2.1 Simdization using vector intrinsics

The first strategy applies short-vector data parallelism (aka SIMD) to the core computational kernel of the sequential program (Figure 2). In this approach, we rewrite the key step in the computation (Figure 2: lines 11–14) using architecture-specific SIMD intrinsics (i.e., SSE on Intel Xeon and VSX on Power7 processors). We store four single-precision float values in a 128-bit SIMD vector variable, and use the corresponding SIMD multiply() and add() intrinsics to implement pair-wise dot-products and additions. Figure 10 shows psuedo-code for the corresponding SIMD kernel. In our SIMD implementation, we use three 128-bit vectors: $\overline{sum}$, $\overline{mask}$, and $\overline{base}$: $\overline{base}$ and $\overline{mask}$ are used to hold elements for four different arrays at the given offset (e.g., base_index, and mask_index) and the `sum` vector is used to store the running sum. As described earlier, the packed `input` array stores elements from the four arrays in four consecutive memory locations, and can be directly indexed for populating the vectors (Figure 10: lines 10-11). For a correlation offset (x, y), the $\overline{sum}$ vector stores and updates four partial sums, which are finally aggregated to compute the resultant `dot_product` value.

4.2.2 Task parallelization

We further parallelize the simdized version. Using this strategy, the tasks of computing correlations are distributed among particpating threads, in an embarrassingly parallel manner. Each thread works on tasks assigned to itself, and update corresponding positions in `correlation_array` without any synchronization. In our implementation, we parallelize the outermost loop (Figure 10: line 1) that traverses the Y-axis of the (250×250) iteration space (in other words, the correlation space gets partitioned in a row-block manner). We implement the task-parallel version using both, the pthreads library and OpenMP.

4.2.3 Evaluation on Intel Xeon X5680 (Westmere)

We ran the original sequential version simdized using SSE4, and a multi-threaded version of the SSE code on a dual six-core Intel Xeon (3.33 GHz X5680) running RHEL 5.4 server. Since each core of the Intel Xeon X5680 supports 2-way SMT, the total number of threads per processor is 12, and the total number of threads per system is 24.

GCC 4.5.0 was used to compile the application. We used both pthreads and OpenMP to implement the multi-threaded version of the simdized code. As the OpenMP environment has better support to manage thread affinity, we report only the OpenMP performance numbers. The GCC compiler was invoked with the flags, `-fopenmp -O3 -std=c99 -msse4 -mtune=native -march=native`. We used the environmental variable, GOMP_CPU_AFFINITY, to control the affinity of OpenMP threads.

The original sequential code ran for 37.26 seconds. The SSE4 version of the code implements the simdized version of the core computation kernel (Figure 10) and executed in 13.60 seconds. This result demonstrated the efficacy of simdizing the innermost step using short-vector data parallelism with 128-bit vector variables. Table 4 presents the results of the OpenMP version of the simdized correlation algorithm on the Intel system. As the results indicate, the performance increased linearly as the number of threads (13.60s to 1.78s). Note that SMT threads are interleaved among the running threads.

4.2.4 Evaluation on IBM Power7

We measure the performance of a manually optimized version using VSX vector intrinsics on a system with two Power7 processors (Table 1) running AIX 6.1. Each Power7 processor contains 8 cores, each supporting 4-way SMT. Hence, each processor can execute upto 32 threads. The latter hand-vectorized code was parallelized with OpenMP. The XL C/C++ compiler (v11.1) was used with flags `-q64 -O5 -qhot -qtune=pwr7 -qarch=pwr7 -qsmp -qthreaded`. Table 5 presents results for the code manually optimized with VSX vector intrinsics and parallelized with OpenMP. Similar to the Intel experiments, we explicitly control thread affinity using environmental variables (e.g., `STRIDE`).

On the Power7 processor, the original sequential code ran for 27.29 seconds. Once vectorized using VSX intrinsics, the same code ran in 11.66 seconds on a single thread (Table 5). As the results indicate, performance improves as the number of OpenMP threads is increased to 32 (11.66s to 1s). As the number of threads was further increased to 64 (across the processor boundary), the

```
0 vector sum, base, mask;
1 for(offset_y=0; offset_y < CORR_SIZE; offset_y++) {
2     for(offset_x=0; offset_x < CORR_SIZE; offset_x++) {
3         Set sum to 0.
4         Set dot_product to 0;
5         Compute correlation_index;
6         for(rows=0; rows < SAMPLE_SIZE-offset_y; rows++) {
7             for(columns=0; columns < SAMPLE_SIZE-offset_x; columns++) {
8                 Compute base_index and mask_index;
9                 base=vector_load(input[base_index]);
10                mask=vector_load(input[mask_index]);
11                sum = vector_sum(sum, vector_mult(base, mask);
12            }
13        }
14        Aggregate sum elements to compute dot_product;
15        correlation_array[correlation_index]+=dot_product;
16    }
17 }
```

Figure 10. Psuedo-code for the SIMD version of the two-dimensional correlation algorithm

Seq	OMP+SSE GCC 4.5.0 #Threads (time in s)												
no SSE	1	2	4	6	8	10	12	14	16	18	20	22	24
37.262	13.601	13.545	7.942	5.548	4.419	3.769	3.492	3.109	2.832	2.518	2.345	2.167	1.786

Table 4. Performance of the sequential and multi-threaded version (using OMP and SSE) of the correlation code on a dual six-core (2-way SMT) Intel Xeon 5680 (Westmere) system using GCC 4.5.0 (time in s)

Seq	OMP+VSX XLC 11.1 #Threads (time in s)									
no VSX	1	2	4	8	16	32	40	48	52	64
27.29	11.66	9.11	6.92	4.15	2.01	1.00	0.911	0.908	0.813	0.650

Table 5. Performance of the sequential and multi-threaded version (using OMP and SSE) of the correlation code on a two processor eight-core Power7 (4-way SMT) system (time in s)

performance further improved, and we observe the best performance of 650ms for 64 threads.

4.3 Analysis of Results

Figure 11 presents the performance of different versions of the correlation algorithm on the NVIDIA Fermi (Tesla 2050), Intel Xeon X5680 (Westmere), and IBM Power7. Figure 11 presents the best performance of GPU versions that use different memory systems for the tile size (250, 2). The texture memory provides the slowest performance, followed by the version that uses the cache hierarchy. The best GPU performance was observed when the shared memory is used exclusively. Results also demonstrate that all Fermi versions of the correlation algorithm outperform versions on both the Xeon X5680 and Power7 systems when a single processor was used in the latter cases. The Intel versions, executing on one processor and on two processors (3.49s and 1.78s), are eight and four times slower than the best GPU implementation (465ms, respectively. These observations are similar to those reported recently from a similar performance study that compared performance of a set of kernels on the Intel Core i7 970 processor and the NVIDIA GTX 280 GPU [3]. The Power7 versions present a different behavior. The single processor version of the Power7 code performs two times slower than the GPU version (1s vs 465ms). However, when the application is executed on two processors, its performance nearly matches the best GPU performance (650ms vs 465ms).

What are the reasons for such a result? The correlation kernel (Figure 2) exhibits no data dependences and can be parallelized easily in an embarrassingly parallel manner over multiple threads. Furthermore, the innermost kernel (lines 11–14, Figure 2) is ideal for parallelization using data-parallel approaches such as simdization. These two features make the correlation application ideal for the Fermi GPU. First, on the Fermi GPU, one can parallelize the correlation algorithm using a very large number of, albeit weaker, threads (i.e., 62,500 threads). Secondly, for each thread, the core correlation kernel can be further optimized by using fused multiply-add (FMA) instructions (Fermi can execute upto 16 simultaneous FMAs per SM). The most important reason for the Fermi performance is the improved memory sub-system. Prior to Fermi, NVIDIA GPUs had 16 KB of shared memory per SM, and no system-managed cache hierarchy. As a result, most of the GPU applications became memory-bound and were not able to harness the available computational horsepower. In contrast, Fermi's larger shared memory and L1/L2 caches enable applications to fully exploit its even larger computing capabilities in a more balanced manner. Table 6 shows the fraction of peak performance sustained on each of three architectures.

As both the Intel X5680 and Power7 have large caches (12 MB L3 on Intel and 32 MB L3 on Power7), the correlation algorithm is compute-bound on the CPUs (input image data just requires 4 MB of space). We believe that on the multi-core CPU side, the compute performance is hampered by two issues: lack of support for wider grain data parallelism, and the lack of a very large number of threads. The importance of data-parallelism for the correlation algorithm is evident from the fact that on both Intel and Power platforms, the performance of the sequential code improved by a factor of 3 (i.e., on Intel: 37.26s to 13.60s, and on Power: 27.29s to

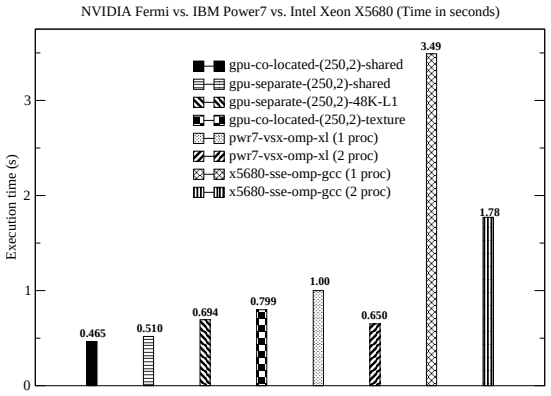


Figure 11. Best performance for different versions of the correlation algorithm

Architecture	Sustained (GFLOPs)	Peak	Sustained %age
NVIDIA Fermi	151.21	1178	12.83%
Intel Xeon X5680	18.06	159.84	11.30%
IBM Power7	70.31	224	31.39%

Table 6. Sustained performance (in GFLOPs)

11.66s) by using 128-bit SIMD vector intrinsics on single-precision floating point values (which translated into 4-way data parallelism). We also used three separate vector intrinsics (`load`, `mult`, and `add`) to implement the multiply and add operation. The second performance issue is related to the number of threads a multi-core CPU execute efficiently? The latest Intel Xeon X5680 processors have 6 cores per chip, where each core can support 2 hyperthreads (2-way SMT). In contrast, the Power7 processor provides 8 cores on a chip, and can support 4-way SMT. Therefore, a dual-core Intel Xeon X5680 system can run upto 24 simultaneous threads, whereas, a dual-core Power7 system can run upto 64 simultaneous threads. For an embarrassingly parallel algorithm like the image correlation, larger number of concurrent threads always helps. This is validated by the results presented in Figure 11, that show that a single Power7 processor using 32 threads requires 1 second, and two Power7 processors using 64 threads requires 650 ms. We believe that with increasing core count per processor, support for wider SIMD, and support for FMAs via vector intrinsics, CPUs would be able to bridge the performance gap.

While it is obvious that GPUs (specifically, the desktop versions) fare much better than CPUs in the performance per price metric, the productivity issues associated with developing and maintaining the GPU code have to be considered as well. As the GPU architecture has evolved over the years, it has become increasingly difficult to optimize code for it. As it is evident from our experimental evaluation, the performance of the GPU code is sensitive to various inter-related factors. In particular, the NVIDIA GPU architecture and the CUDA programming model have brought the knotted integrated thread scheduling and tiling problem to the forefront. For example, once the thread mapping is finalized, all further memory optimizations, e.g., identifying the optimal tile volume, must be made within the constraints of the chosen thread map. If a different thread map is selected, one needs to start again. In contrast, on multi-core CPUs, parallelization via OpenMP or pthreads, and memory optimizations are relatively independent and complementary aspects. With the advent of the cache sub-system on the NVIDIA Fermi, the program optimization problem has become even more convoluted. Recall that on Fermi, performance in the space of memory access optimization for the same correlation kernel varied from 465ms to 12.52s! Though past works have highlighted the importance of using shared memory on GPUs [1, 8], we have presented a more in-depth analysis of the complexity of this memory hierarchy through a particular case. For our problem, we spent several weeks optimizing code for the NVIDIA Fermi system. In contrast, on the CPU systems, it took us a few hours to develop optimized parallel versions from the original sequential code. Using well-known parallelization techniques like simdization and parallelization via OpenMP or pthreads, we were able to reduce the GPU performance gains drastically (i.e., from 37.26s/27.29s vs. 465 ms, to 650 ms/1.78s vs 465ms). We believe the multi-core versions could be further optimized. In addition, the multi-threaded versions of the code on CPUs were incremental with respect to the original sequential code. Our experience thus presents compelling evidence that, when considering performance with productivity and code reusability, CPUs still have a clear edge over GPUs.

5. Conclusions

We evaluated the performance of a real-world image processing application on the NVIDIA Fermi and state-of-the-art commodity multi-core processors using a wide variety of parallelization techniques. An OpenMP-based version of the application running on a dual-processor Power7 system was able to approach the GPU performance. This is a compelling productivity result, given the effort required to develop an equivalent high-performance CUDA implementation. Our experiences in developing the GPU version have highlighted the importance of optimizing sufficiently for various types of memories in order to achieve the best performance. In future, we plan to perform detailed analysis of the memory behavior of modern GPU processors such as the NVIDIA Fermi and also explore automatic ways for tuning GPU applications.

6. Acknowledgments

We thank Damir Jamsek, Raja Das, and Brian Hall for providing us access to the Fermi and Power7 systems.

References

[1] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, Feb. 2008.

[2] Khronos Group. OpenCL - the open standard for parallel programming of heterogeneous systems.

[3] V. W. Lee, C. kim, J. Chhugani, D. M, D. Kim, A. Nguyen, M. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammerlund, R. Singhal, and P. Dubey. Debunking the 100x gpu vs cpu myth: an evaluation of throughput computing on cpu and gpu. In *Proc. 37th International Symposium on Computer Architecture (ISCA)*, June 2010.

[4] NVIDIA Inc. NVIDIA CUDA programming guide, version 3.0.

[5] NVIDIA Inc. NVIDIA's next generation CUDA compute architecture: Fermi.

[6] A. Olmos and F. A. Kingdom. *McGill Calibrated Colour Image Database*. 2004. http://tabby.vision.mcgill.ca.

[7] A. R. Rao and B. Schunck. Computing oriented texture fields. *Computer Vision, Graphics and Image Processing: Graphical Models and Image processing*, 53(2):157–185, 1991.

[8] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82, 2008.