**RC24982 (W1004-095) April 23, 2010**
**Computer Science**

# IBM Research Report

## Believe it or Not! Multi-core CPUs Can Match GPU Performance for FLOP-intensive Application!|

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

# Believe It or Not! Multi-core CPUs can Match GPU Performance for a FLOP-intensive Application!

Rajesh Bordawekar Uday Bondhugula Ravi Rao

IBM T.J. Watson Research Center, Yorktown Heights, New York

E-mail: {`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* $O(n^4)$ *operations involving dot-products and additions. We implement this algorithm on a nVidia GTX 285 GPU using CUDA, and also parallelize it for the Intel Xeon (Nehalem) and IBM Power7 processors, using both manual and automatic techniques. Pthreads and OpenMP with SSE and VSX vector intrinsics are used for the manually parallelized version, while a state-of-the-art optimization framework based on the polyhedral model is used for automatic compiler parallelization and optimization. The performance of this algorithm on the nVidia GPU suffers from: (1) a smaller shared memory, (2) unaligned device memory access patterns, (3) expensive atomic operations, and (4) weaker single-thread performance. On commodity multi-core processors, the application dataset is small enough to fit in caches, and when parallelized using a combination of task and short-vector data parallelism (via SSE/VSX) or through fully automatic optimization from the compiler, the application matches or beats the performance of the GPU version. The primary reasons for better multi-core performance include larger and faster caches, higher clock frequency, higher on-chip memory bandwidth, and better compiler optimization and support for parallelization. The best performing* *versions on the Power7, Nehalem, and GTX 285 run in 1.02s, 1.82s, and 1.75s, respectively. These results conclusively demonstrate that, under certain conditions, it is possible for a FLOP-intensive structured application running on a multi-core processor to match or even beat the performance of an equivalent GPU version.*

## 1 Introduction

One of the key 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 4 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 and OpenCL 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.

Therefore, one of the key questions to be addressed is whether a FLOP-intensive scientific application should be parallelized on a multi-core CPU or accelerated on a GPU. This paper presents 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 $O(n^4)$ computation over 2-dimensional single-precision floating point matrices, where each step involves 4 multiplications and 4 additions. Our results illustrate that on a nVidia GTX 285 GPU, this application suffers from increased accesses to slower off-chip device memory due to smaller shared memory, and unaligned device memory accesses, expensive atomic operations to support collective aggregation, and weaker single-threaded performance. Due to nVidia GPU's architectural constraints, an inherently compute-intensive application such as this becomes memory-bound. In contrast, on a multi-core CPU, the application's data set can fit into its cache, and the CPU's hardware capabilities and compiler support can together exploit the application's inherent parallelism effectively. Our results illustrate that performance of the cross-correlation application optimized on a latest nVidia GPU can be matched by hand-tuned or compiler-optimized versions of the application executing on the Intel Xeon (Nehalem) or IBM Power7 processors. The best performing versions on the Power7, Nehalem, and GTX 285 run in 1.02s, 1.82s, and 1.75s, respectively. The application when optimized with advanced compiler auto-parallelization technology ran in 1.67s on a single Power7 processor, still beating the GPU version. In addition, a significantly higher amount of effort was put into developing the GPU implementation when compared to CPU ones; the compiler-optimized code was generated fully automatically. This paper helps understand the reasons behind these results.

The intent of this work is not to claim that the performance of any GPU can be matched by a multi-core processor[1]. Our results identify certain scenarios where performance of an application running on the nVidia GPU can be matched by its multi-core counterparts. Our observations regarding the GPU performance are not restricted to the application under evaluation or to the image processing domain. They are applicable to all those problems that require global collective reduction operations (that require usage of atomic instructions), cannot use shared memory for storing the input or intermediate data, and generate accesses that are difficult to be optimized via hardware coalescing.

[1]The results are restricted to nVidia GPUs only

For example, we have also observed similar issues with the GPU implementation of relational grouping and aggregation operators (e.g., SQL `GROUP BY` and `COUNT`) on large data sets; in this case, multiple GPU threads perform atomic updates on random locations in a large hash table stored on the device memory.

The rest of the paper is organized as follows. Section 2 discusses the application. Section 3 describes its acceleration on the nVidia 285 GPU. Section 4 discusses how our compiler framework automatically parallelizes and optimizes it, while Section 5 describes hand optimization through vector intrinsics and task parallelism. Finally, results are presented in Section 6.

## 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 $(x_1, x_2)$. We also use a 2-dimensional matrix consisting of the 2-element tuples $(a, b)$, which represents the color in the $a - b$ plane [9]. 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 in Section 2, 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 $500 \times 500$ 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 $500 \times 500$ using single-precision floating point numbers. Thus, total space consumption per image is 4 MB. The result of the correlation is a $250 \times 250$ single-precision matrix, consuming 250 KB of space.

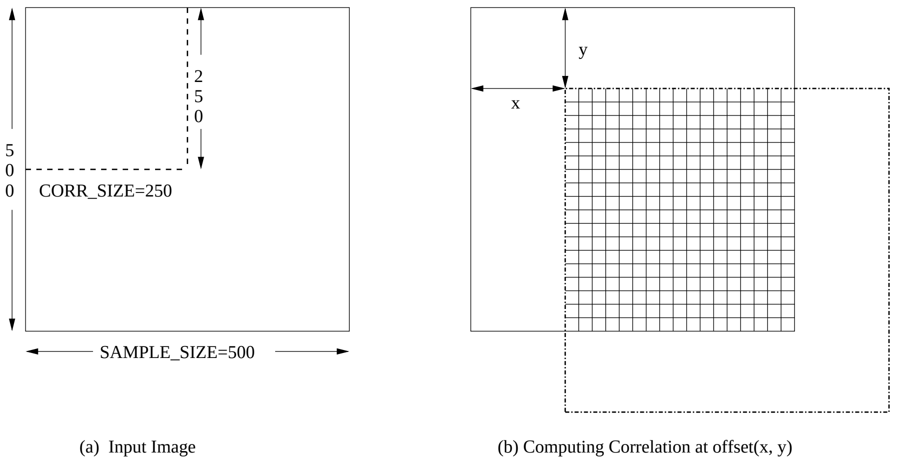

**Figure 1. Computing 2-dimensional correlation**

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 $250 \times 250$ (62500) positions. For each position in the correlation space, the overlapped sections of the base image, and its shifted version, called the mask, are correlated (shaded region in Figure 1(b)). Figure 2 presents the corresponding psuedo-code. The correlation computation is implemented using 4-level nested loops over 4 arrays: `color1`, `color2`, `orientation1`, and `orientation2`. The outer-most

```
1 for(offset_y=0; offset_y < CORR_SIZE; offset_y++) {
2     for(offset_x=0; offset_x < CORR_SIZE; offset_x++) {
3         base_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                  base_index = (offset_y + rows)*SAMPLE_SIZE + offset_x + columns;
9                  mask_index = (offset_y + rows)*SAMPLE_SIZE + columns;
10                 correlation_index = offset_y*SAMPLE_SIZE + offset_x;
11                 dot_product = (color1[base_index]*color1[mask_index])+
12                 (color2[base_index]*color2[mask_index])+
13                 orientation1[base_index]*orientation1[mask_index])+
14                 orientation2[base_index]*orientation2[mask_index]);
15                 correlation_array[correlation_index]+=dot_product;
16             }
17         }
18     }
19 }
```

**Figure 2. Psuedo-code for the two-dimensional correlation algorithm**

two loops correspond to shifts in the two-dimensional correlation space, while the inner-most two loops correspond to computation over individual data sets. In every step, two elements of each array are fetched with two different offsets: `base_index` and `mask_index`. The correlation operation implements a pair-wise dot-product over aligned pair of values from base and mask from overlapped sections of every data set and adds the computed dot-product values to determine the final correlated value for that shift (Figure 2, lines 11–15). In a parallel environment, this corresponds to a collective reduction operation over $(O(n^2))$ data.

The overall computation requires $O(n^4)$ 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 $(250 \times 250 \times O(n^2))$ additions (when parallelized, $250 \times 250$ collective reduction operations, each adding $O(n^2)$ dot-products) for computing the final result. The rest of the paper focuses on evaluating different strategies for parallelizing this correlation kernel.

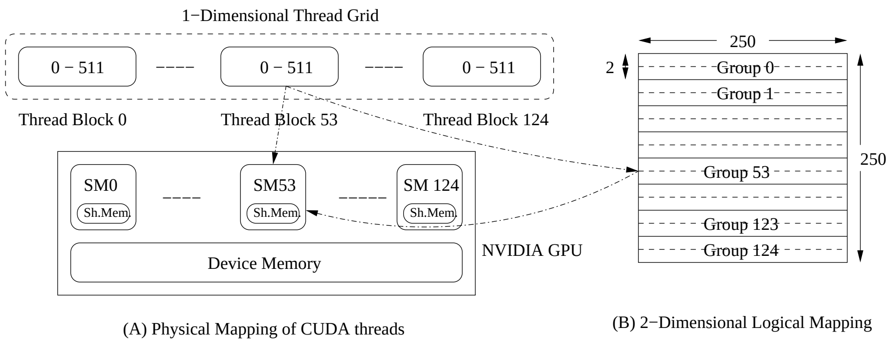

**Figure 3. Thread mapping in the CUDA implementation**

## 3 GPU Implementation

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 result space, (250, 250), is smaller than the input data space, (500, 500). Therefore, if we were to use $500 \times 500$ threads, with one thread per element in the input 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 $250 \times 250$ threads to parallelize the correlation computation (Figure 3: A). Each thread would be busy throughout the application, but each thread would perform a different amount of work. The $250 \times 250$ threads are allocated as a 1-dimensional grid of 125 thread blocks, each with 512 threads. Each thread block is assigned to a separate symmetric multiprocessor; thus 512 processors share the 16 KB shared memory. Logically, the $250 \times 250$ threads are viewed as a 2-dimensional grid, in which each thread block corresponds to a thread group with 250 columns and 2 rows (Figure 3: B). Each thread is assigned a logical ID in the logical $(250, 250)$ 2-dimensional space.

The next step is to determine the optimal memory layout for the correlation algorithm. The algorithm reads each image using four data sets, each representing a $500 \times 500$ single precision float array. Thus, the total memory usage per image is 4 MB. The result is stored in a $250 \times 250$ single-precision float array which consumes 250 KB of space. Neither the input arrays nor the result array can be stored in the shared memory. Each step of the correlation algorithm performs two read accesses per each array (e.g., `color1`) (Figure 2: lines 11–14); thus the memory requirement of each correlation step is 32 bytes. The two accesses to an array are strided by an offset determined by the correlation shift. Such accesses cannot be coalesced. Considering the mapping of $250 \times 250$ threads into a logical two-dimensional $250 \times 250$ grid, the memory footprint of a $2 \times 250$ thread group is $500 \times 32$ bytes (16 KB). In practice, shared memory on a GPU stores data less than 16 KB of data. Thus, the shared memory region in a symmetric multiprocessor cannot store even the data consumed by threads in a thread group for a single step in the correlation algorithm. We run into the same problem even if we use $500 \times 500$ threads. Hence, we store both the input data arrays and the result array on GPU device memory. It should be noted that the proposed thread-mapping enables cross-thread hardware coalescing while accessing individual arrays. However, strided accesses to individual arrays cannot be optimized using hardware coalescing. Accesses to individual arrays can be further improved by packing four single-precision float arrays into a single large `float4` array: the $i^{th}$ `float4` element of the packed array consists of $i^{th}$ elements of the four input arrays. Thus, a single memory access to a packed array can return a `float4` element that contains four single-precision float elements from four different arrays. Finally, since the inout data arrays are read-only, they can be accessed using GPU's texture memory. Thus, for the correlation algorithm, we have four different memory access alternatives based on separate and packed arrays, each accessed using either texture memory or direct memory.

Given the thread mapping and memory layout, 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 $(x, y)$. The shared and global aggregation approaches produce numerically incorrect results as CUDA currently supports atomic operations only for integers. We still describe them for comparison with the local aggregation solution. Figure 4 illustrates computation of correlation at offset $(73, 109)$ using the three different implementations.

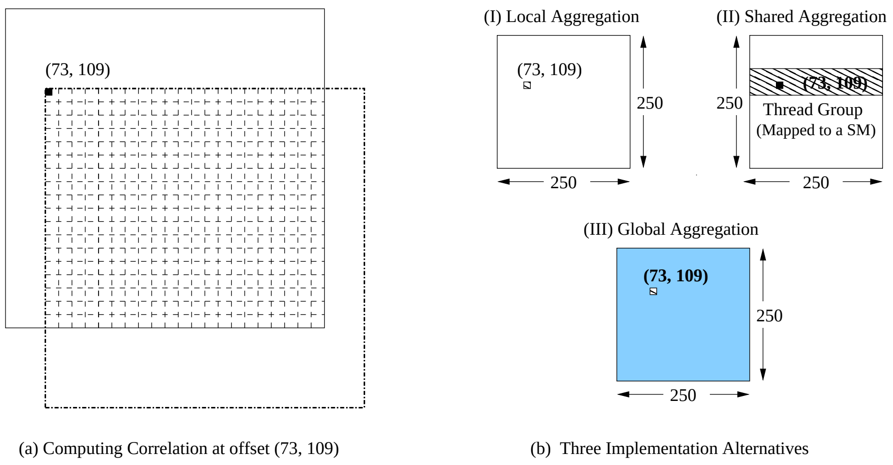

**Figure 4. Three alternative CUDA implementations. The shaded regions in (b) represent the processors involved in computing the correlation at the offset (73,109). The shared and global aggregation approaches produce numerically incorrect results.**

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 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, where all threads concurrently compute results for their corresponding offsets, without the need for any coordination on global or shared memory variables.

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). Therefore, the collective addition operation is implemented using a variable stored in the shared memory of a symmetric multiprocessor. 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). This approach improves over the local aggregation approach on two aspects: memory accesses to fetch data sets and corresponding computation are made in parallel. However, CUDA 3.0 supports the `atomicAdd()` function only for integer values. Hence, we cast the float values as integers in the `atomicAdd()` function. Thus, the final result would be numerically incorrect[2].

The final implementation approach further improves the parallelism employed in the correlation process. In the global aggregation approach, computation of each offset in the correlation space is implemented using all $250 \times 250$ 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. Finally, like the shared aggregation approach, this approach computes numerically incorrect values.

[2]One can use atomic compare-and-swap operations to implement floating-point atomic additions. But this approach would be slower than the current one as it involves busy waiting.

## 4 CPU Implementation: Compiler-driven auto-parallelization

The compiler we use to study this aspect is IBM XL [4], IBM's compiler for C, C++, and Fortran, targeting IBM microprocessor architectures. It includes the polyhedral compiler framework as one of its passes as part of its high-level loop optimization phase. The polyhedral framework applies a long sequence of transformations on a mathematical representation of the intermediate language. Transformations include affine transformations encompassing permutation, skewing, fusion, cache/register tiling, with extraction of parallelism. In the rest of this section, we describe how the polyhedral framework in the XL compiler optimizes the application. Most optimizations described below are either known or state-of-the-art; however, as we will see, some of these are applicable to our application in a context completely different from what has been previously explored. We now discuss the transformations performed by the compiler for parallelization and simdization.

### 4.1 Parallelization

The core computation involved is a 4-dimensional loop nest as was shown in Figure 2. Before this code can be further analyzed in a polyhedral representation, one should note the way traditional compiler passes transform it. The code in its original form does not appear to be affine at all. However, after propagation of constants and copies, all array accesses are indeed affine. Data accessed in the core computation is via pointers passed to the procedure containing the loop nest. Hence, without some amount of inter-procedural analysis, it would not be known whether these pointers alias. At the -O5 optimization level of the XL compiler, all its interprocedural data flow and alias analysis is enabled. When the polyhedral pass is called at this level, we are able to view these as array accesses as well as disambiguate between these for dependence testing purposes. Hence, the entire nest is extracted for analysis and transformation in the polyhedral representation.

The algorithm driving loop transformation in our polyhedral pass is aimed at reducing dependence distances hierarchically, bringing parallelism to outer levels while keeping reuse at inner levels [1].

In addition, maximal *bands* of permutable loops are detected to maximize cache and register tiling opportunity as well as reason about certain late transformations to enable loop vectorization or further improve spatial reuse. A *band* of loops is a consecutive set of loops that is fully permutable. For this code, the core affine transformation chosen by the framework is, trivially, the identity one with two such bands. Properties of its loops are as follows:

| | Figure 2 Line # | Property | Band |
|---|---|---|---|
| Loop 1 | 1 | parallel | band 0 |
| Loop 2 | 2 | parallel | band 0 |
| Loop 3 | 6 | fwd dep | band 0 |
| Loop 4 | 7 | seq | band 1 |

The outer two loops are detected as parallel, which is also not surprising to the programmer. The inner two loops perform a reduction. The third loop has all dependence components in the forward direction along it, and so the outer three loops form a single permutable band, i.e., a 3-d blocking can be performed on those. All four dimensions are not identified as being blockable here due to the presence of a reduction on the innermost two loops, which strictly speaking, is serializing if one does not relax non-associativity of floating-point additions. Assuming floating-point additions to be associative would lead to certain dependences being discarded here, and allow the code to be inferred as 4-d blockable. However, in this case, we do not perform any cache tiling due to data sets completely fitting in caches. The obvious choice for parallelizing the nest is the one parallelizing the outermost loop as it provides the coarsest granularity of synchronization-free parallelism with no other downsides with respect to how data would be shared between processors. Hence, auto-parallelization of this loop nest is fairly straightforward.

## 4.2 Simdization

SIMD parallelism [3, 8, 5] when extracted from loops is a form of fine-grained parallelism that obeys certain constraints with respect to accessed data. Data accessed by simdized loop iterations has to be aligned and contiguous. The amount of parallelism is limited to the width of vectors, at least from the compiler's code generation viewpoint. Hence, SIMD parallelism can be readily extracted from any parallel loop that also satisfies data alignment and contiguity constraints. Misaligned accesses can be handled through various compile-time or runtime techniques involving shifting, shuffling, and/or code versioning [3, 8]. In some cases, these can be done efficiently, while in others, its impact can completely negate the benefits of SIMD. In addition to innermost loops that are parallel, outer loops can be vectorized too. Outer loop vectorization [5] is a technique through which a few iterations of a non-innermost loop are moved all the way inside and the resulting short trip count loop is vectorized. The number of iterations brought is the same as the SIMD width. Hence, a parallel loop at any particular position, not necessarily innermost, can be simdized subject to data contiguity constraints.

Our application presents an interesting puzzle for simdization owing to its data access pattern. Consider the accesses on the RHS of the code. The arrays are subscripted by:

$$\begin{aligned} \text{base\_index}: &\quad (offset\_y + i) * \text{SAMPLE\_SIZE} + offset\_x + j \\ \text{mask\_index}: &\quad (offset\_y + i) * \text{SAMPLE\_SIZE} + j \end{aligned}$$

Note that there are two ways loop vectorization can be performed here. One can perform outer loop vectorization on loop `offset_x`, while reduction vectorization can be performed along loop $j$. However, the `base_index` subscript is the problematic one. Irrespective of which loop is simdized, all four accesses using `base_index` would be misaligned whenever `offset_x` is not a multiple of four. Accesses using `mask_index` do not suffer from this issue. We find that performing loop simdization here with misaligned accesses leads to a significant degradation in performance: this is in spite of special techniques to reduce alignment overhead. However, a much simpler and effective simdization technique can be applied here. It involves extracting simd parallelism from a single iteration of the loop nest. Such simdization can only be performed here after a particular data layout transformation, array interleaving, described below.

**Array Interleaving:** Transformations that convert a structure of arrays to an array of structures or those that interleave arrays to improve locality have been studied [2, 10, 11]. Array interleaving (also known as array regrouping) interleaves a group of arrays so that elements at a particular index of each array are contiguous in the new data structure. If the same index is used to access each array, say in the same iteration, regrouping the array can (1) improve spatial reuse, (2) reduce the number of prefetch stream buffers, and (3) enable basic block simdization. Enabling vectorization through array interleaving has not been previously reported or utilized to the best of our knowledge. Array interleaving is same as the array packing employed in the GPU implementation. However, the compiler simdization framework has limited support for basic block simdization and in this case is unable to perform it. We perform this optimization for the version optimized manually, as described in the next section.

## 5 CPU Implementation: Hand Optimization

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

```
vector $\overline{sum}$, $\overline{base}$, $\overline{mask}$;
for(offset_y=0; offset_y < CORR_SIZE; offset_y++) {
    for(offset_x=0; offset_x < CORR_SIZE; offset_x++) {
        Set $\overline{sum}$ to 0.
        Set dot_product to 0;
        Compute correlation_index;
        for(rows=0; rows < SAMPLE_SIZE-offset_y; rows++) {
            for(columns=0; columns < SAMPLE_SIZE-offset_x; columns++) {
                Compute base_index and mask_index;
                $\overline{base}$=vector_load(input[base_index]);
                $\overline{mask}$=vector_load(input[mask_index]);
                $\overline{sum}$ = vector_sum($\overline{sum}$, vector_mult($\overline{base}$, $\overline{mask}$);
            }
        }
        Aggregate $\overline{sum}$ elements to compute dot_product;
        correlation_array[correlation_index]+=dot_product;
    }
}
```

**Figure 5. Psuedo-code for the SIMD version of the two-dimensional correlation algorithm**

**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 5 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 5: 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.

**Task Parallelization** We further parallelize the simdized version. Using this strategy, the tasks of computing correlations are distributed among participating 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 5: line 1) that traverses the Y-axis of the $(250 \times 250)$ 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.

## 6 Experimental evaluation

**Experimental setup** 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 $500 \times 500$ 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.

| | Processor architectures | | |
|---|---|---|---|
| | nVidia GTX 285 | Intel Xeon E5570 | IBM Power7 |
| Cores | 30×8 Symmetric Processors | 4 cores with 2-way SMT | 8 cores with 4-way SMT |
| Frequency | 1.476 GHz | 2.93 GHz | 3.55 GHz |
| Memory | 16 KB Shared Memory per SM<br>1 GB Device Memory | 32 KB L1 I and D caches<br>256 KB L2 cache per core<br>8 MB Shared L3 cache | 32 KB L1 I and D caches<br>256 KB L2 cache per core<br>32 MB L3 cache (eDRAM) |
| Memory bandwidth | 159 GB/s | 22.89 GB/s | 100 GB/s (sustained) |
| SIMD | | 2 64-bit FMA or 4 32-bit FMA | 4 32/64-bit FMAs |

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

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.

**Evaluation on nVidia GTX285** We have evaluated the three CUDA versions of the correlation algorithm on a nVidia GTX 285 GPU hosted on a dual quad-core Intel Xeon (Nehalem) MacPro system running MacOS 10.6. As discussed in Section 3, the shared and global aggregation versions of the code produce incorrect results. Hence, we will provide detailed experimental results for the local aggregation code, and discuss the remaining two alternatives only for performance comparison.

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

Among the three different CUDA versions, the local aggregation approach takes the least amount of time, and the global aggregation code takes the most. On a sample image, the local aggregation code required 1.72 seconds, the shared aggregation code took 2.58 seconds, and the global aggregation code took more than 10.5 seconds! The key reason for the performance degradation is the cost of atomic operations to implement collective operations on the nVidia GPU. *For every correlation offset*, the global aggregation strategy invokes 500*125 atomic locks on a shared memory variable and 125 atomic locks on a device memory variable. In contrast, *for every iteration offset*, the shared aggregation strategy uses 500 atomic locks on a shared memory variable. The local aggregation strategy does not use any atomic locks, hence performs the best among the three alternatives.

| | Packed array (time in s) | | Separate arrays (time in s) | |
|---|---|---|---|---|
| Image | texture | direct | texture | direct |
| 0 | 1.75 | 5.43 | 3.66 | 2.84 |

**Table 2. Performance comparison of different memory access patterns for the local aggregation strategy (time in s)**

Table 2 presents evaluation of the local aggregation strategy using different memory configurations. Referring back to the code fragment presented in Figure 2, each step makes 8 memory accesses, 2 accesses each to the four arrays. The two accesses to an array use distinct varying offsets, and cannot be coalesced. To optimize memory accesses, we packed the four arrays and accessed them by treating the packed elements as `float4` values from a single array. Table 2 presents results of the correlation algorithm for the packed and separated array configurations, using both direct and texture memory accesses. The results demonstrate that: (1) The execution time for the local aggregation application varies from 1.75 sec to 5.42 sec depending on how data is accessed (i.e., the application is memory bound), and (2) Packed and separated memory accesses respond differently to texture and direct memory accesses. Texture memory works the best for packed memory accesses and direct memory accesses to separated arrays provide the highest performance. While accessing packed arrays directly, each consecutive access is separated by a 16-byte offset, and cannot be optimized via hardware coalescing (the number of coalesced accesses to the device memory went up by 4 when packed arrays were used.) Similarly, when multiple arrays are accessed via texture memory, the texture cache gets thrashed due to contention among fetched data blocks (on a nVidia GPU, the texture memory fetches data into a cache in a blocked fashion[3].) Each step results in 8 separate

[3]The texture memory layout has not been publicized. It has been guessed that data in the texture memory is laid out using some form of space-filling curve. The texture memory accesses are also not reported in the CUDA profiler.

memory accesses, resulting in fetching 8 blocks, and leading to cache thrashing. For packed texture and separated direct memory accesses, we further explored unrolling of the innermost loop (Figure 2: lines 7–15); unrolling produced little or no improvement for the separated direct memory scenario and for the packed texture case, the performance degraded. The performance numbers presented in this section do not include any unrolling.

These results demonstrate that on a nVidia GTX 285 GPU, the best performance of the correlation application (1.725 sec) is achieved using the local aggregation strategy that accesses input data as packed arrays via texture memory.

**Evaluation on Intel Xeon** We ran the original sequential version SIMDized using SSE4, and a pthreads version of the SSE code on a dual (2-way SMP) quad-core Intel Xeon (2.93 GHz X5570) running RHEL 5.3 server. Since each core of the Intel Xeon supports 2-way SMT, the total number of threads per chip is 8, and the total number of threads per system is 16. We used the GCC version 4.3.4 compiler to build the application. The compiler is invoked with the requisite optimization flags (e.g., -O3, -mtune, -march, -funroll-loops etc.). We found the OpenMP support in gcc 4.3.4. to be problematic. Hence, we only present pthread performance numbers. We expect the OpenMP performance to be similar to that of the pthreads version.

| | Seq | Pthreads+SSE #Threads (time in s) | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Image | no SSE | 1 | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
| 0 | 30.17 | 12.900 | 7.482 | 3.944 | 2.646 | 2.188 | 2.245 | 1.984 | 1.901 | 1.875 |

**Table 3. Performance of the multi-threaded version of the correlation code on a dual quad-core Intel Xeon(Nehalem) system using Pthreads (time in s)**

The original sequential code, compiled with gcc, ran for 30.17 seconds. The SSE4 version of the code implements the simdized version of the core computation kernel (Figure 5) and executed in 12.89 seconds. This result demonstrated the efficacy of simdizing the innermost step using short-vector data parallelism with 128-bit vector variables. Table 3 presents the results of a multi-threaded version of the simdized correlation algorithm on the Intel system. We found that the OpenMP support in gcc 4.3.4 does not work properly on the Intel Xeon architecture and hence, we only present numbers from experiments that use the pthreads threading library. Results illustrate that the performance increases linearly as the number of threads is increased to 8 (12.9 sec to 2.64 sec), after 8 threads (across the chip boundary) the performance improves slightly, and we observe the best performance of 1.86 seconds for 16 threads.

| Seq-packed (no VSX) | 14.27 | | | | |
|---|---|---|---|---|---|
| | #Threads | | | | |
| | 1 | 4 | 8 | 8×2 | 8×4 |
| Manual (VSX+OpenMP) | 12.55 | 3.76 | 1.79 | 1.38 | 1.02 |
| Compiler (XL) | 14.37 | 4.51 | 2.37 | 2.92 | 2.15 |
| Compiler (XL/Poly) | 14.37 | 3.67 | 1.84 | 2.24 | 1.69 |

**Table 4. Performance of hand-parallelized and compiler-driven auto-parallelized versions of the correlation code on a single Power7 processor (time in s)**

**Evaluation on Power7** We measure the performance of the original code optimized and parallelized by the compiler and that of a manually optimized version using VSX vector intrinsics on a system with a single Power7 processor (Table 1) running AIX 6.1. 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', with an additional flag to enable the polyhedral pass in case of 'XL/Poly'. Table 4 presents results for: (1) code manually optimized with VSX vector intrinsics and parallelized with OpenMP, (2) auto-parallelized by the IBM XL compiler without the polyhedral pass, and (3) auto-parallelized by the IBM XL compiler with the polyhedral pass. 8×2 and 8×4 in the table corresponding to two and four SMT threads on each of the eight cores, respectively. We see linear scalability in all cases. As mentioned in Section 4, the compiler is unable to auto-simdize this code. However, even without simdization, we see that the compiler generated code performs nearly as well as the hand-simdized code. One would have expected better single-thread performance for the hand-vectorized code, but we believe this was not the case because of the manually vectorized and OpenMP parallelized code not being as well optimized by the backend (and other lower-level optimization passes in the compiler) as the original unmodified code. This is also strongly supported by the fact that SMT (8×2, 8×4) provides significant improvement for the hand-vectorized code, but not for the compiler optimized versions, an indicator of poor ILP exhibited by the former. Though unrolling and register tiling could have been explored in conjunction with hand-vectorization, we did not perform this study as SMT seemed to have satisfactorily recovered these losses to a good extent. In future, we would like to find ways in which a compiler can automatically simdize applications such as this one efficiently: this will provide even higher performance, mostly likely beating the hand-vectorized version.

## 6.1 Analysis of Results

The results show the nVidia GPU 285 implementation of the correlation algorithm being outperformed by versions on the Power7 processor. The Power7 implementations, both the hand-optimized, and the one optimized by the compiler, provide improvements of 18% and 3% over the best GPU implementation. Similarly, performance of the application on a dual quad-core Xeon (2-way SMT) system matches GPU performance. In addition, the results also highlight the amount of performance that can be achieved by the compiler fully automatically with absolutely no effort on part of the programmer to parallelize or simdize the code.

What are the reasons for such a result? On the GTX 285, due to the smaller shared memory, the application was forced to keep all its data in the device memory. Further, to avoid expensive atomic operations in the collective reduction step, the application used individual threads in an embarrassingly parallel manner. Weak single-thread performance and un-optimized device memory accesses transformed a numerically-intensive application into a memory-bound one. In contrast, the CPUs can hold the application data in their caches, and exploit available software and hardware resources to effectively parallelize the embarrassingly parallel application (e.g., on the Power7 processor, the application performance improved from 14.27 seconds to 1.02 seconds using VSX and OpenMP). Note that the correlation application was not constrained by device memory size and compute resources (GTX 285 can host many more threads than $250 \times 250$). In summary, the application performance was affected by key architectural features of nVidia GPUs.

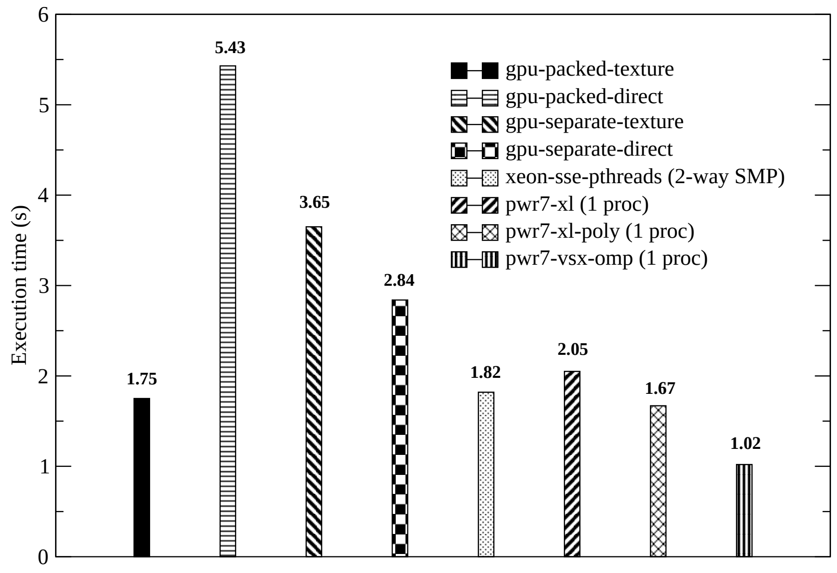

**Figure 6. Best performance for different versions of the correlation algorithm**

## 7 Conclusions

We evaluated the performance of a real-world image processing application on the latest GPU and commodity multi-core processors using a wide variety of parallelization techniques. A pthreads-based version of the application running on a dual quad-core Intel Xeon system was able to match nVidia 285 GPU performance. Using fully automatic compiler-driven auto-parallelization and optimization, a single Power7 processor was able to achieve performance better than that on the nVidia 285 GPU. This is a compelling productivity result, given the effort required to develop an equivalent high-performance CUDA implementation. Our results also conclusively demonstrate that, under certain conditions, it is possible for a program running on a multi-core processor to match or even beat the performance of an equivalent GPU program, even for a FLOP-intensive structured application. In future, we plan to compare performance of such applications on upcoming GPU architectures from AMD and nVidia, e.g., nVidia Fermi.

## Acknowledgments

We would like to acknowledge Tommy Wong, Salem Derisavi, Tong Chen, and Alexandre Eichenberger for useful comments and help in understanding some performance anomalies.

## References

[1] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *International conference on Compiler Construction (CC)*, Apr. 2008.

[2] C. Ding and K. Kennedy. Inter-array data regrouping. In *12th International Workshop on Languages and Compilers for Parallel Computing*, 1999.

[3] A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI*, pages 82–93, 2004.

[4] IBM XL Compilers. http://www.ibm.com/software/awdtools/xlcpp.

[5] D. Nuzman and A. Zaks. Outer-loop vectorization: revisited for short simd architectures. In *Parallel architectures and compilation techniques*, pages 2–11, 2008.

[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] P. Wu, A. E. Eichenberger, and A. Wang. Efficient simd code generation for runtime alignment and length conversion. In *CGO*, pages 153–164, 2005.

[9] G. Wyszecki and W. Stiles. *Color Science: Concepts and Methods, Quantitative Data and Formulae*. Wiley, 1982.

[10] P. Zhao, S. Cui, Y. Gao, R. Silvera, and J. N. Amaral. *orma*: A framework for safe automatic array reshaping. *ACM Trans. Program. Lang. Syst.*, 30(1), 2007.

[11] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN PLDI*, pages 255–266, 2004.