Parallel Computing 39 (2013) 79–93

Contents lists available at SciVerse ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

All-pairs computations on many-core graphics processors

Abhinav Sarje [a,*], Srinivas Aluru [b]

[a] Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
[b] Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA

ARTICLE INFO

Article history:
Received 19 September 2011
Received in revised form 28 December 2012
Accepted 4 January 2013
Available online 23 January 2013

Keywords:
Parallelization
High-performance
GPU
GPGPU
Multicores

ABSTRACT

Developing high-performance applications on emerging multi- and many-core architectures requires efficient mapping techniques and architecture-specific tuning methodologies to realize performance closer to their peak compute capability and memory bandwidth. In this paper, we develop architecture-aware methods to accelerate all-pairs computations on many-core graphics processors. Pairwise computations occur frequently in numerous application areas in scientific computing. While they appear easy to parallelize due to the independence of computing each pairwise interaction from all others, development of techniques to address multi-layered memory hierarchies, mapping within the restrictions imposed by the small and low-latency on-chip memories, striking the right balanced between concurrency, reuse and memory traffic etc., are crucial to obtain high-performance. We present a hierarchical decomposition scheme for GPUs based on decomposition of the output matrix and input data. We demonstrate that a careful tuning of the involved set of decomposition parameters is essential to achieve high efficiency on the GPUs. We also compare the performance of our strategies with an implementation on the STI Cell processor as well as multi-core CPU parallelizations using OpenMP and Intel Threading Building Blocks.

© 2013 Elsevier B.V. All rights reserved.

1. Introduction

Emerging multi-core and many-core processors are increasingly being used in data and compute-intensive applications to obtain high-performance. Efficient parallelization techniques and software engineering are essential in order to harness the raw computational potential these architectures offer. Pairwise computations occur in numerous scientific applications across many areas, ranging from molecular dynamics and many-body simulations [1] to systems biology [2] and clustering algorithms [3]. While specifics vary, all these applications involve all-pairs computations between n entities, each described by a d-dimensional vector. Typically, n and/or d can be large, making acceleration of such computations critical to achieve high-performance and enable large-scale applications.

Other than its use in direct form, all-pairs computation, in many cases, forms a part of a more complex algorithmic strategy. The Fast Multipole Method [4] is an example of such an algorithm, where all-pairs computations are restricted to certain "neighborhoods" in a larger scheme of approximations based on a hierarchy of spatial decomposition. While such algorithmic strategies should be used whenever possible, the run-times are often dominated or co-dominated by all-pairs computations.

In this paper, we study the problem of all-pairs computations, generalizing it by assuming a computational kernel $\mathcal{F}$ is given depending on the application. Pairwise computations have been studied previously on multi-core architectures in

* Corresponding author. Tel.: +1 5104952793.
E-mail addresses: asarje@lbl.gov (A. Sarje), aluru@iastate.edu (S. Aluru).

0167-8191/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.parco.2013.01.002

the context of a specific application that the researchers are trying to solve, including on the Cell processor [5–8], and graphics processors [9,10]. As the problem arises frequently in many contexts and efficient parallelization rarely depends on the specifics of the application, we consider this problem in its abstract form, and develop common architecture-aware algorithmic strategies to extract maximum performance. In earlier work, we developed efficient schemes for this problem on the Cell processor [21,11,5]. In this paper, we focus on developing methods for all-pairs computations on many-core graphics processors.

We describe the generalized all-pairs computations problem in Section 2. Given the fine-grain parallelism offered by graphics processors, we propose a hierarchical decomposition scheme in Section 3 to perform all-pairs computations efficiently on graphics processors taking into account the memory hierarchy. In Section 4 we provide an in-depth analysis of how performance varies as a function of the various decomposition parameter values. Many GPU implementations overlook this issue. We show that a clever choice of these parameter values is essential to achieving maximum performance, in some cases improving the performance by an order of magnitude, and demonstrate how to tune the values of these parameters given a GPU architecture. We review our decomposition scheme for the coarse-grained thread parallelism offered by the STI Cell processors in Section 5 and use this implementation to compare the performance of our schemes on graphics processors. Further, we also compare GPU performance with parallel implementations using OpenMP and Intel Threading Building Blocks on general purpose multi-core CPUs in Section 6.

2. Generalized pairwise computations

The problem of all-pairs computation can be abstracted as follows: Given two input matrices M_1 and M_2, of sizes $n_1 \times d$ and $n_2 \times d$, respectively, compute an output matrix D of size $n_1 \times n_2$ where $D[i,j] = \mathcal{F}(M_1[i, 0 \ldots (d-1)], M_2[j, 0 \ldots (d-1)])$. Here, $\mathcal{F}$ is a computational kernel function, and $M_k[i, 0 \ldots (d-1)] = \langle M_k[i,0], M_k[i,1], \ldots, M_k[i,d-1] \rangle$ represents a d-dimensional vector. See Fig. 1 for an illustration. In general, n_1, n_2 and d can be arbitrary, and $\mathcal{F}$ can be any binary function.

Computing D requires $n_1 \cdot n_2$ evaluations of the function $\mathcal{F}$. In many applications, the computational complexity of $\mathcal{F}$ is $O(d)$, making the overall computational complexity of constructing D to be $O(n_1 n_2 d) = O(n^2 d)$, assuming $n_1 = O(n)$ and $n_2 = O(n)$.

If memory access speeds and memory size limitations are not an issue, parallelizing all-pairs computation is trivial because each entry in D can be computed independently. Such an obvious parallelism often extends to even the computation of $\mathcal{F}$, useful when the dimensionality is very large. However, it becomes more challenging on specialized processors like graphics processors and the Cell processor where architecture-aware techniques are essential in harnessing the raw computational power these architectures have to offer. Such emerging architectures provide multiple levels of parallelism, as well as a memory hierarchy – from a large but slow off-chip memory to fast but small on-chip memory. The Cell processor has a limited Local Store (256 KB) residing on each of the main computational cores (SPEs) and all memory management is done by a user explicitly. On current graphics processors, each multiprocessor (SM) has a small shared memory (~16 KB, 48 KB, depending on architecture) which is shared by a set of threads assigned to that multiprocessor. The use of shared memory is done explicitly. A clever use of these small on-chip memories, as well as memory transfers, are the key to obtain high performance on such architectures for memory bound problems like the one we consider. In the following we present efficient and architecture-aware strategies on GPUs, and compare performance with Cell and multi-core CPU implementations.

3. Developing an efficient scheme on GPUs

We develop our schemes for all-pairs computation based on the NVIDIA GPGPU and the CUDA programming model [12]. GPUs provide fine-grained parallelism, with potentially thousands of threads running simultaneously. The basic architecture of an NVIDIA GPU consists of an array of Streaming Multiprocessors (SM), or simply multiprocessors, where each SM consists of a number of scalar processors (SP), such as eight in older GPU chips and 32 in Fermi architectures. A small on-chip shared memory is available on each SM, accessible only to the threads running on that particular SM and shared among threads in a

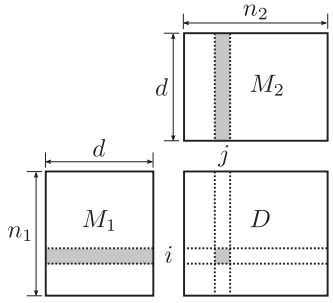

Fig. 1. Generalized all-pairs computation: two input matrices, each containing d-dimensional vectors. Output D is constructed by applying kernel function $\mathcal{F}$ on each pair of vectors (i,j) taken from M_1 and M_2.

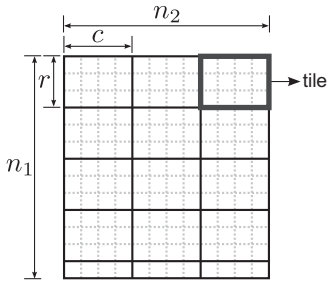

Fig. 2. Computations in D are decomposed into tiles, each tile representing a sub-matrix with r rows and c columns.

"thread-block" (described below). This memory has a very low access latency and a high bandwidth and provides light-weight synchronization for use among the corresponding threads. Each thread running on the GPU also has access to a large off-chip memory, with the cost of a high latency. A GPU is not a general purpose processor, and hence requires a CPU along with it. The CUDA programming model provides simple abstractions over the GPU architecture. It is based on a hierarchy of groups of threads. The CPU runs a host program, and launches kernels on the GPU device. Computations are specified into independent *CUDA thread blocks*, each of which is either a $1,2$ or 3-dimensional array of threads. A thread block is executed on an SM independent of other thread blocks. All the thread blocks together define a $1,2$ or 3-dimensional *grid*. A kernel launch by the host program, hence, executes this grid of thread blocks on the GPU device.

The computation of each output entry $D[i,j]$ in the all-pairs problem is independent of the computation of other entries. This independence provides an advantage for making efficient use of the fine-grained parallelism offered by GPUs. Therefore, to compute the matrix D on a GPU, a decomposition of D into *tiles* will work well (Fig. 2). Each tile represents a sub-matrix D_{tile} of D, containing r rows and c columns. Such a 2-D decomposition performs well on the GPU model by providing better cache locality which we will describe later. Also, a decomposition along both dimensions gives more flexibility on the number of input vectors. For example, a 1-D decomposition would require a portion of vectors from one input matrix, but also whole of the second matrix, which may be infeasible for large sizes. In our tiling scheme, a CUDA thread block is responsible for computing all entries in a tile. For this case, we define a thread block to be of the same size as a tile, $r \times c$, where each entry in the tile is computed by a separate thread in the thread block.

Although this naive decomposition takes advantage of the high degree of parallelism offered by GPUs, it is not efficient. For the computation of each entry $D[i,j]$, a thread reads the two input vectors $M_1[i,0\ldots(d-1)]$ and $M_2[j,0\ldots(d-1)]$ directly from the high latency off-chip device memory. A vector from M_1 (M_2) is read n_2 (n_1) times from the device memory. In the following, we present techniques to enhance this basic tile decomposition, by making use of the memory hierarchy to obtain an efficient implementation on a GPU.

3.1. Efficient all-pairs computation on a GPU

To compute D efficiently on a GPU, we need to make explicit use of the fast on-chip shared memory. We focus on two principal factors essential for high-performance: minimizing number of memory load operations, and maximizing data reuse. For now, let us assume that the dimensionality of the input vectors d is small enough to enable multiple vectors to fit in the small shared memory available on an SM. We will relax this assumption later. To compute a tile, we require r row input vectors and c column input vectors. Therefore, first, the threads in a thread block collectively load these row and column vectors into the shared memory. The size of the required row vectors is $r \times d$. To efficiently load them into the shared memory by the corresponding $r \times c$ threads, each thread loads a single dimension of one of the vectors, thereby collectively loading an $r \times c$ block of data at a time. This minimizes the total number of required loads. Hence, $\lceil \frac{d}{c} \rceil$ number of such loads are performed (Fig. 3).

Once the row and column vectors corresponding to a tile D_{tile} are loaded into the shared memory, thread (i,j) computes a single entry $D_{tile}[i,j]$. Hence, a row (column) vector is reused from the shared memory by all the threads responsible to com-

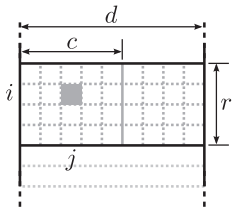

Fig. 3. Loading of input vectors corresponding to a tile into the shared memory is performed simultaneously by threads in a block. Shown here is scheme for loading r row vectors from the input matrix M_1, in chunks of c dimensions at a time, where each thread loads a single dimension. Thread (i,j) loads the element (i,j) in the block. A total of $\lceil \frac{d}{c} \rceil$ transfers is performed by each thread. Similar scheme is followed for loading the c column vectors.

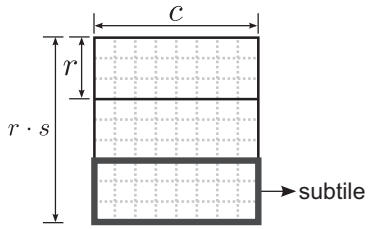

Fig. 4. Decomposition of a tile into subtiles to enable further reuse of column vectors once they are loaded into the shared memory. A tile is computed by the corresponding CUDA thread block, one subtile at a time. Shown here is a single tile. With s subtiles, there are $r \cdot s$ rows in a tile.

pute the corresponding row (column) in D_{tile}. Note that since the life time of the shared memory is the same as that of the corresponding block, we can no longer reuse these vectors further for computations of other tiles.

To enable added reuse of vectors (either rows or columns) already loaded in the shared memory, we introduce a further decomposition: A tile is decomposed into *s subtiles*, where the size of a subtile is now $r \times c$, resulting in the size of a tile to be $(r \cdot s) \times c$. A CUDA thread block, of size $r \times c$, is responsible to compute a whole tile, one subtile after another. This way, once the column vectors are loaded into the shared memory at the beginning of the processing of a tile, the same are reused for all subsequent subtiles in the tile, while only row vectors need to be loaded. Note that only one dimensional subtiling is beneficial since when moving from one subtile to the next, either row or column vectors may remain the same, but not both. This subtiling scheme is demonstrated in Fig. 4. Subtiling enables further reuse of vectors at the cost of bringing sequentiality in the processing, since the same thread block computes all subtiles in a tile one after other, but may improve performance with a proper choice of the number of subtiles in a tile, as we will see later in our experiments.

3.2. Generalizing to higher dimensions

In the above we assumed that d is small enough for multiple vectors to fit in the shared memory on an SM. In cases where d is too large for the vectors to fit as a whole in the shared memory, we decompose the input vectors into *slices*, each vector slice containing d' ($\leqslant d$) dimensions. This results in a total of $\left\lceil \frac{d}{d'} \right\rceil$ slices of the input vectors. We process each slice, one after the other, following the tiling and subtiling scheme described above. Note that processing of a slice is independent of all other slices, obviating the need to look for any possibility of reusing vectors between slices. Processing of each slice generates partial results in D which may be stored in the global memory. These can be then aggregated after all partial results are available. This scheme is shown in Fig. 5.

We point out that the slicing decomposition will be valid only if the function $\mathcal{F}$ consists of associative operations that can be carried out independently, while non-slicing version can take any kernel function. As a simple example of a decomposable kernel function, consider the decomposition of the scalar product of two vectors $a = \langle a_0, \ldots, a_{d-1} \rangle$ and $b = \langle b_0, \ldots, b_{d-1} \rangle$, which forms the kernel function of matrix–matrix multiplication operation:

$$\mathcal{F}(a,b) = \sum_{i=0}^{d-1} a_i \cdot b_i \tag{1}$$

$$= \sum_{j=1}^{\left\lceil \frac{d}{d'} \right\rceil} \left(\sum_{i=d' \cdot (j-1)}^{\min(d, d' \cdot j)-1} a_i \cdot b_i \right). \tag{2}$$

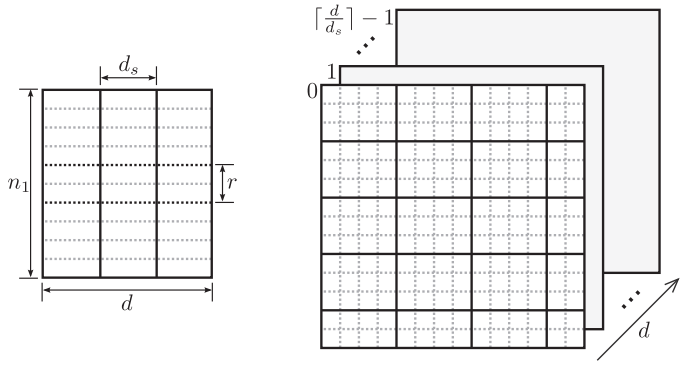

Fig. 5. Decomposition of input vectors in M_1 into slices (left), each containing d_s dimensions. Similar decomposition is done for vectors in M_2. Partial results of the output matrix are generated from each slice computation (right). Corresponding slice numbers are indicated above. Final D is obtained by a reduction of the partial results.

Another example of such a decomposable kernel is the L_p-norm distance metric used in many scientific applications, for instance, similarity computations in microstructures of heterogeneous materials [13], and coherent structures discovery in fluid dynamics:

$$\mathcal{F}(a,b) = \left(\sum_{i=0}^{d-1} |a_i - b_i|^p\right)^{\frac{1}{p}} \tag{3}$$

$$= \left(\sum_{j=1}^{\left\lceil \frac{d}{d'} \right\rceil} \left(\sum_{i=d'\cdot(j-1)}^{\min(d,d'\cdot j)-1} |a_i - b_i|^p\right)\right)^{\frac{1}{p}}. \tag{4}$$

In the above decomposition of the function $\mathcal{F}$, the inner summation represents the computational kernel for a single slice, and the outer summation (and power function in L_p-norm metric) represents the partial result aggregation.

In many applications, the kernel function $\mathcal{F}$ is trivially decomposable, for instance Spearman's rank correlation [14], and in some others it may require more complex algorithmic strategies, possibly with auxiliary storage requirement. This applies, for instance, to compute Pearson correlations, and Mutual Informations using B-spline estimator [15] or kernel estimators [16]. Hence, this slicing approach is applicable to many real-life problems.

4. Analyzing performance on GPU

Decomposition of input vectors and output computations into slices, tiles and subtiles, raises the question of choosing optimal values for the various parameters: r, c, s and d', to obtain highest possible performance. In this section, we address this question, leveraging on the various architectural features and constraints of a generic NVIDIA GPU.

In the following, we conduct experiments on two different NVIDIA GPU architectures. The first platform is a 2.0 GHz quadcore Intel Xeon (Nehalem architecture based) system equipped with NVIDIA Quadro FX 5800 graphics processor. This GPU consists of 30 SMs, each with 8 SPs, giving a total of 240 SPs. Each SM provides 16 KB of on-chip shared memory and 16 K registers, which are shared among the threads residing in the SM. This GPU card has 4 GB off-chip DDR3 device memory. The CUDA compute capability of this device is 1.3, and has CUDA driver 3.0 installed.

The second system consists of a 2.93 GHz hexcore Intel Xeon (Nehalem) CPU equipped with an NVIDIA M2090 (Fermi architecture based) graphics processor. This GPU consists of 16 SMs, each with 32 SPs, giving a total of 512 SPs. Each SM houses a configurable 64 KB memory, which we configure to 48 KB shared memory and 16 KB L1-cache. Each thread block has access to 32 K registers. There is also an L2 cache of size 768 KB. This GPU has a 5~ GB off-chip DDR3 device memory. The CUDA compute capability of this GPU is 2.0, and has CUDA driver 4.2 installed.

4.1. Computational kernel

We implemented generalized all-pairs computation using CUDA as a generic library to accelerate such computations. This library provides an intuitive C/C++ interface while hiding all complexities of the GPU implementation, and supports both single and double precision computations. We implemented the L_p-norm distance metric (Eq. 4) as an example of a computational kernel with our general all-pairs computation scheme. Note that our scheme/library is not limited to this and any other kernels can be easily plugged-into replace this L_p-norm kernel, e.g. a dot-product kernel will realize matrix multiplication.

Pairwise L_p-norm distance computations occur in many scientific applications. Two particular cases we are interested in come from the areas of fluid dynamics, and materials science. One of the key factors in the design of Flap-Wing Micro Air Vehicles (MAVs) is the aerodynamics of their flapping-wings. Computational fluid dynamics simulations of such MAV designs provide raw data about the fluid pressures, created due to the wind movement, at various points in the space around the wings [17]. L_p-norm distance metric is employed to analyze such raw data in order to identify coherent structures by obtaining the change in fluid pressure at various points with time. A time snapshot of the simulation gives a vector in d dimensions, where each dimension represents a point in space, and hundreds or thousands of such snapshots are taken to form a matrix M with n d-dimensional vectors. The first data set we use is obtained from such MAV simulations, and has $d = 5419$. In another application, pairwise L_p-norm is employed to construct stochastic models of a given set of properties of a set of microstructure samples, obtained from heterogeneous media [13]. One such sample is given as a vector in d dimensions, where each dimension represents a pixel in the image of that sample. The second data set we use corresponds to this application, and it has $d = 40{,}000$. In both data sets, we vary the number of vectors n with the experiments.

4.2. Features and constraints

As described previously, in the CUDA programming model, computations on a thread block are performed independently of all other thread blocks in the grid, and each thread block is assigned to a SM for execution. Following the Single-Instruction Multiple-Threads (SIMT) architecture, an SM manages and executes threads from a block in groups called warps. Having multiple warps scheduled on an SM is beneficial for performance since it hides instruction and memory latencies during

the execution. Furthermore, if there are p SMs on a GPU, we want the total number of thread blocks in the grid to be well in excess of p. A large number of thread blocks ensures a better scheduling and load balance among the SMs. Therefore, we term the total number of thread blocks in the grid as *concurrency*, P. Following our tile and subtile decomposition scheme, a tile is assigned to one thread block, hence, the concurrency would be:

$$P = \left\lceil \frac{n_1}{r \cdot s} \right\rceil \cdot \left\lceil \frac{n_2}{c} \right\rceil \tag{5}$$

For the purpose of analysis below, and simplicity, we represent P as $\frac{n^2}{r \cdot s \cdot c}$ (assuming $n_1 = n_2 = \Theta(n)$).

The limited amount of on-chip shared memory m available on an SM is divided among all the thread blocks assigned to that SM. Shared memory usage for a single tile is $(r + c) \cdot d' \cdot b_i$, where b_i is the size used to encode a single input dimension (e.g. 4 bytes in single precision, 8 bytes in double precision, or single precision complex numbers, and so on). The amount of shared memory usage puts a constraint on the number of thread blocks that can be scheduled on an SM simultaneously. The maximum number of thread blocks that can be scheduled on an SM is approximately equal to $\left\lfloor \frac{m}{(r+c) \cdot d' \cdot b_i} \right\rfloor = O\left(\frac{m}{(r+c) \cdot d'}\right)$. A similar constraint is also established by the number of registers on an SM, which are partitioned among the corresponding warps.

Contiguous memory accesses by threads in a warp from the device memory enables memory coalescing, reducing the memory transactions to be carried out, and thereby, improving the performance. When vector slices are loaded into the shared memory, for a single vector slice, c dimensions are loaded by c threads at once. Since this memory access is contiguous (following row-wise storage of input matrices), these accesses are coalesced. Therefore, a larger c enables further performance improvement. CUDA also puts forth limitations on the maximum thread block size that can be created, disallowing large values for r and c.

The number of SMs on a chip vary with various NVIDIA GPUs. Their number has both increased as well as decreased over the various generations of NVIDIA GPUs. On the other hand, the number of SPs per SM have remained the same in the past, but are increasing with the GPU generations: NVIDIA Fermi architecture has 32 SPs per SM [18], the future Kepler architecture will have even more SPs). Furthermore, the size of on-chip memory available per SM has also been increasing. Taking into account these trends in GPU architecture development, based on our experiments, we analyze the effect of varying the parameter values on performance on the two GPU platforms described previously, and address the question of optimizing performance with a good choice of these parameters. In the following, we represent the four parameters as a 4-tuple: $\langle c, r, s, d' \rangle$.

4.3. Varying thread block/ subtile size $r \times c$

A proper choice of a thread block size, and its dimensions are essential to obtain high performance on GPUs. The degree of concurrency ($n^2/r \cdot s \cdot c$) can be increased by decreasing c and r. Scheduling enough warps on an SM ensures better resource usage by hiding instruction latencies. On the other hand, a better reuse of input vectors loaded into the shared memory is achieved by increasing c and r, thereby reducing device memory access latency. Therefore, we need to strike the right balance between the two extremes for the thread block sizes.

To analyze the effect of varying the number of rows and columns in a block on performance, we use our implementation to conduct the following experiments: in the first set of experiments, we obtained the performance for varying c while keeping other parameter values constant. To gain a better insight, we defined a small set of different parameter configurations (e.g. $\langle c, 8, 4, 50 \rangle$), where for each configuration, c is variable while others are kept constant. In our set of configurations, we use different values of the other parameters to see the effect of each of them, while the remaining two are held constant. We used two data sets, with varying n and d: 2000×5419 and $1000 \times 40{,}000$. We ran the experiments on the two aforementioned GPU platforms to see how the architectural differences affect performance. The variations in runtime for this set of experiments are shown in Fig. 6.

Low values of c (< 8) perform the worst due to two main factors: the number of warps in a thread block is small, and the device memory accesses cannot be efficiently coalesced in a warp. For each of the configurations, we see a spike at certain c values (e.g. c = 8, 16, etc.), which are more significant for the FX5800 architecture than for Fermi M2090. This is mainly because of two reasons: the warp size versus number of threads available, and the shared memory usage per block limiting the number of blocks scheduled on an SM, and these spikes show such transition points. We note that the configurations with small block sizes (r values) perform the worst among those shown: For small blocks, the number of threads in a thread block are not be enough to fill a warp size thereby wasting resources, while for higher c values, the row vector re-usage r would be lesser.

Among the shown configurations with different number of subtiles values, we observe that a moderate value of s ($\sim$20) performs good overall, while $s = 1$ performs good for small c values since the column vector reuse does not play an important role in these cases. Furthermore, when n is larger (Fig. 6 top), large values of r and c have a higher performance. This is because as long as there is enough overall concurrency available, larger c values ($\geqslant$16) enable higher input vector reuse when s is also large. A similar trend is seen for the different slice size configurations. Therefore, taking all these factors together, when enough shared memory is available, such as on M2090 compared to FX5800, and there is enough concurrency (larger n) in the problem, a larger c value contributes to better performance.

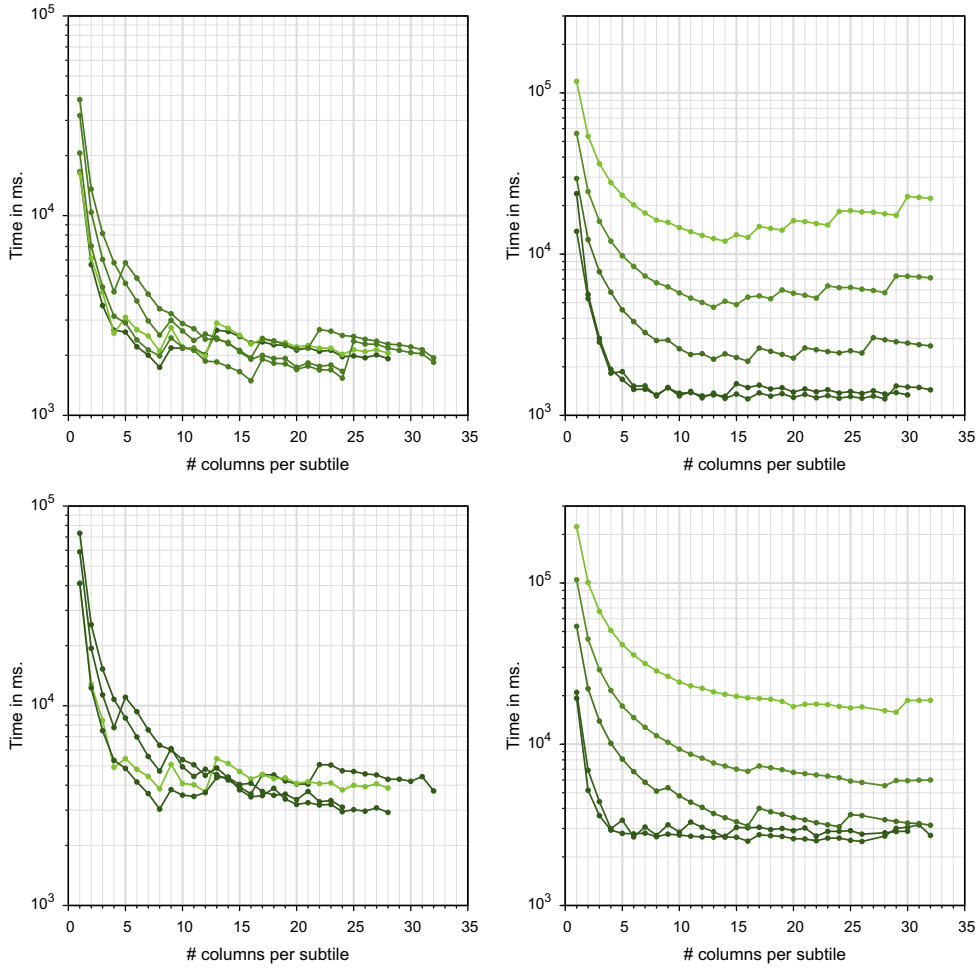

Fig. 6. Varying c values for various parameter configurations. Input sizes are n_1=n_2=2000, d = 5419 (top plots), and n_1=n_2=1000, d = 40,000 (bottom plots) on two Nvidia GPUs: FX5800 (left plots) and M2090 (right plots). Lighter shades represent smaller block sizes.

A balanced value of number of rows in a subtile, r, is also essential to the performance. We conducted similar experiments for varying r values, and the corresponding graphs are shown in Fig. 7. In general, we observe similar trends as seen for varying c values. We see that the cases with $c > r$ tend to perform better. This is because c vectors are reused for all the subtiles in a tile, while r vectors are reloaded for each subtile, and for a given block size ($r \cdot c$), higher c enables higher vector reuse. The configurations with various s values perform similar for the cases when $r > 16$, because with larger r, the latencies involved in loading r row vectors balance the gain from the reuse of smaller c vectors. These observations also reinforce our previous statement that for better reuse of column vectors, $c > r$ is helpful. Not shown in the figures, we also confirmed that the case with $c = 16, r = 1$ outperforms the case with $c = 8, r = 1$ by a factor of 1.5. Overall, we see that a value of $r = 8$ performs the best, with the effect being less on the newer Fermi architecture attributed to larger shared memory size.

From these experiments, we observe that a proper choice of c and r is crucial for high-performance and can improve the performance by over an order of magnitude. Once a high-performing subtile size is chosen, further performance improvement tuning can be done by a choice of the remaining two parameters: s and d', which we discuss next.

4.4. Number of subtiles s in a tile

A large s value (number of subtiles in a tile) contributes to a higher reuse of the same input column vectors in a given tile. On the other hand, since the subtiles are processed one after the other in a sequential manner, a larger s value increases the tile size, thereby reducing the concurrency available in the computations (see Eq. 5). Further, with less concurrency, and larger tile sizes, load balance among the multiprocessors on the GPU also suffers. Hence, a balanced value for s is desirable for optimal performance. We conduct experiments with varying values of number of subtiles, s, for different configurations of parameters. The corresponding data are shown in the graphs in Fig. 8. Note that varying s does not alter the amount of shared memory and register usage, hence the behavior on the two GPU architectures is similar on this regard.

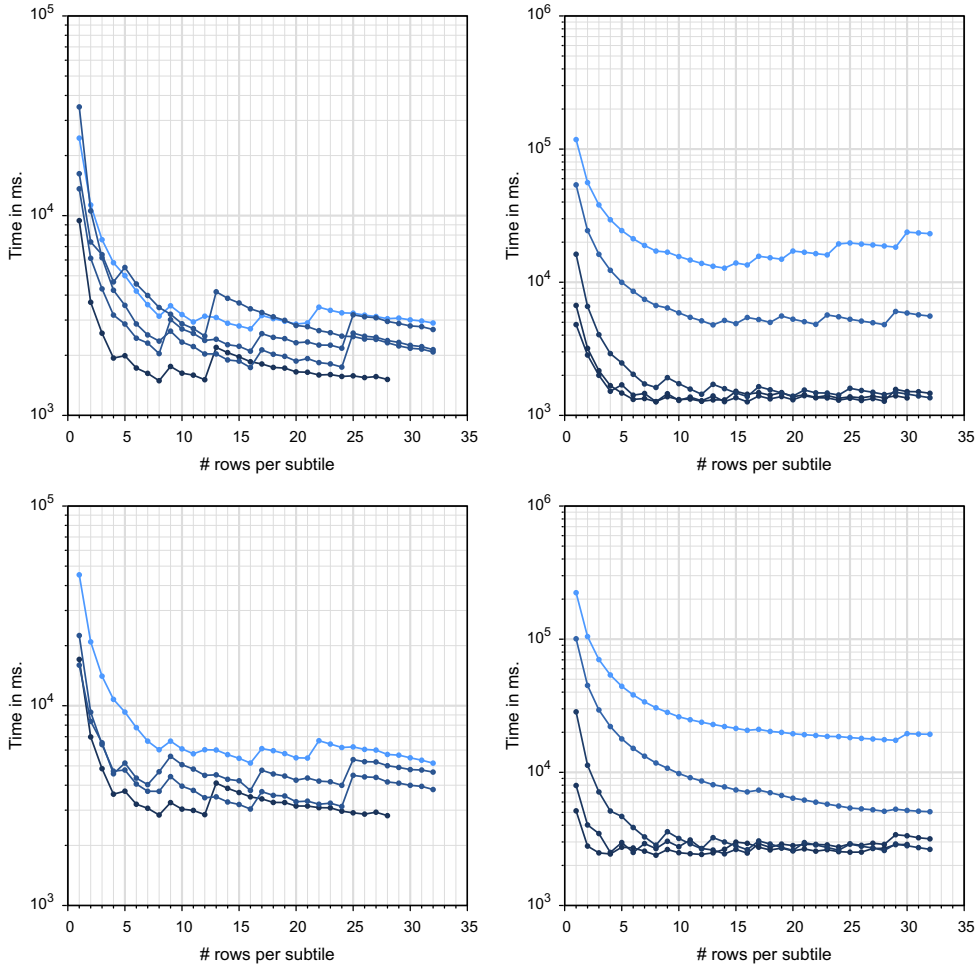

Fig. 7. Varying r values. Input sizes are $n_1 = n_2 = 2000, d = 5419$ (top plots), and $n_1 = n_2 = 1000$, $d = 40{,}000$ (bottom plots) on two Nvidia GPUs: FX5800 (left plots) and M2090 (right plots). Lighter shared represent smaller block sizes.

In the graphs shown, certain values of s have spikes where the performance is higher compared to neighboring smaller s values. These spikes occur when the grid size is reduced by one due to increasing tile sizes, thereby reducing resource wastage. This is more prominent on FX5800 than M2090 since the former GPU has 30 SMs, while latter has 16. For a particular grid size, the performance gradually decreases owing to increased resource wastage. We observe this behavior since with larger number of SMs, higher concurrency available in the problem is beneficial for performance. We also observe that for small values of c, there is almost no significant performance improvement beyond $s \sim 5$ on FX5800 and $s \sim 20$ on M2090; this is because the reuse of c vectors is low. For the configurations where block sizes are small there is a significant improvement when s increases from 1 to 2 because in the former case, the overall number of input vector loads from the device memory is higher and each block has less data to be reused, making $s > 1$ more beneficial in hiding the memory latencies.

Notice that when the number of input vectors is increased (Fig. 8 bottom), the graphs flatten out, with this behavior being more prominent on M2090 than on FX5800. This is because increasing s reduces the concurrency and load balance among the SMs available. Hence, for a larger input on the same GPU architecture, a larger s value will still ensure enough concurrency while increasing column vector reuse, while on different GPUs, the number of SMs available dictates the values of s.

4.5. Input vectors slice size d'

The size of a vector slice determines how many total slices need to be processed. Each slice generates partial results which are stored into the device memory. Hence, a large number of slices (small d') contributes to a reduced performance due to larger number of writes to the high latency device memory. On the other hand, large values of d' increase resource usage on the SMs, reducing the number of warps that can be simultaneously scheduled. Hence, the value of d' should also be carefully

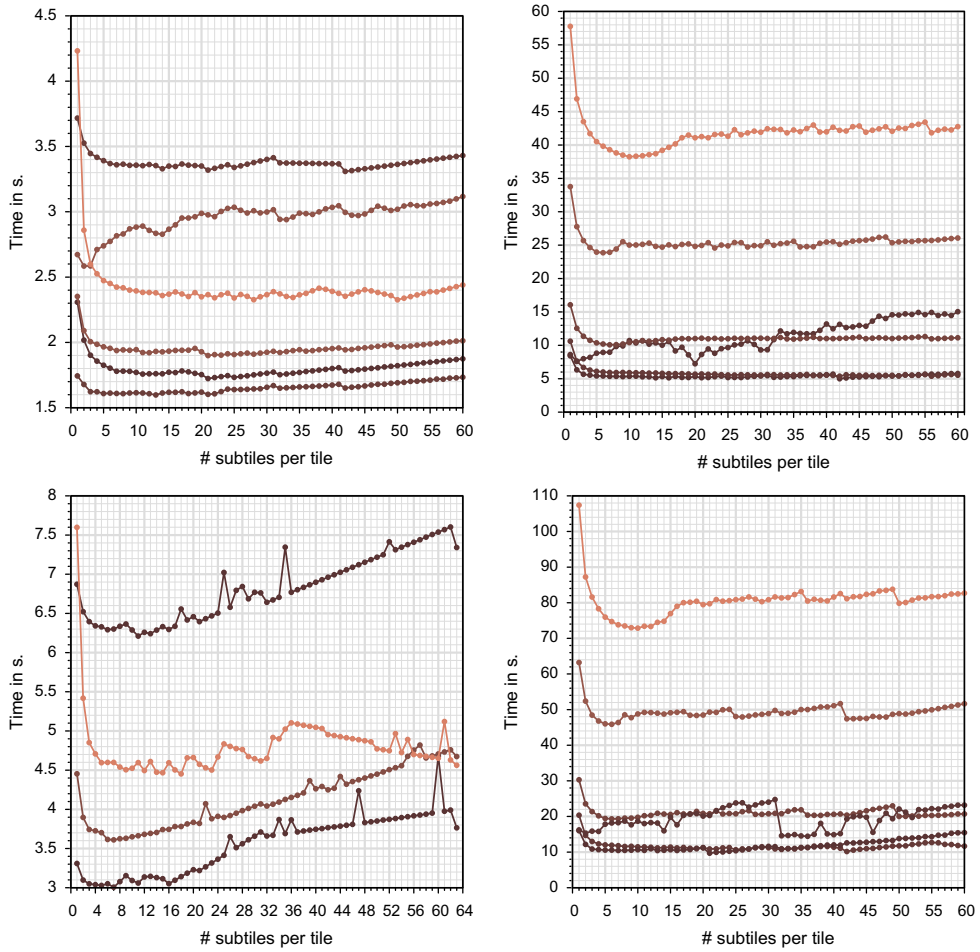

Fig. 8. Varying s values for various parameter configurations. Input sizes are $n_1 = n_2 = 2000, d = 5419$ (top plots), and $n_1 = n_2 = 1000$, $d = 40{,}000$ (bottom plots) on Nvidia FX5800 (left plots) and M2090 (right plots). Lighter shades represent smaller block sizes.

balanced to maximize performance. Results of our experiments with varying values of d' for various parameter configurations are shown in Fig. 9.

The performance difference is observed to be significantly different on the two GPU architectures for this parameter variation, although on both platforms, for all the configurations, we notice a periodic pattern in the execution times – they gradually decrease, and then spike before decreasing again. These spikes are created because the input vector slices are loaded c dimensions at a time simultaneously by the thread block, requiring a total of $\lceil \frac{d'}{c} \rceil$ loads by each thread. When d' value increases, it decreases the number of idle threads for the last transfer (and also, reduces memory coalescing) till the whole thread block contributes to the collective memory loading. Increasing it further increases the number of transfers, resulting in a sudden decrease in performance.

One of the major factors contributing to the difference in behavior on the two GPUs is the peak bandwidth difference, which is higher on M2090, because each slice computation requires result to be written to the main device memory, and higher number of slices would tend to perform worse. We also note that when d' is large, the configurations with small c and large r perform the worst on FX5800. This is because with increasing d', the amount of input data loaded from device memory increases. Hence, a large r results in higher latencies, which overshadows the gain from reuse of small number of c column vectors. In these experiments, we see that an intermediate value of d' ($\sim$50) has the best performance for most configurations on this GPU.

4.6. Choosing the parameter values

With the possible values of each parameter, we need to find the best choice from a large search space. Instead of performing an exhaustive search for each experiment, below we provide some guidelines for the choices based on our experimental observations and reasonings. From the extensive experiments performed, we see that for the NVIDIA Quadro FX 5800 graph-

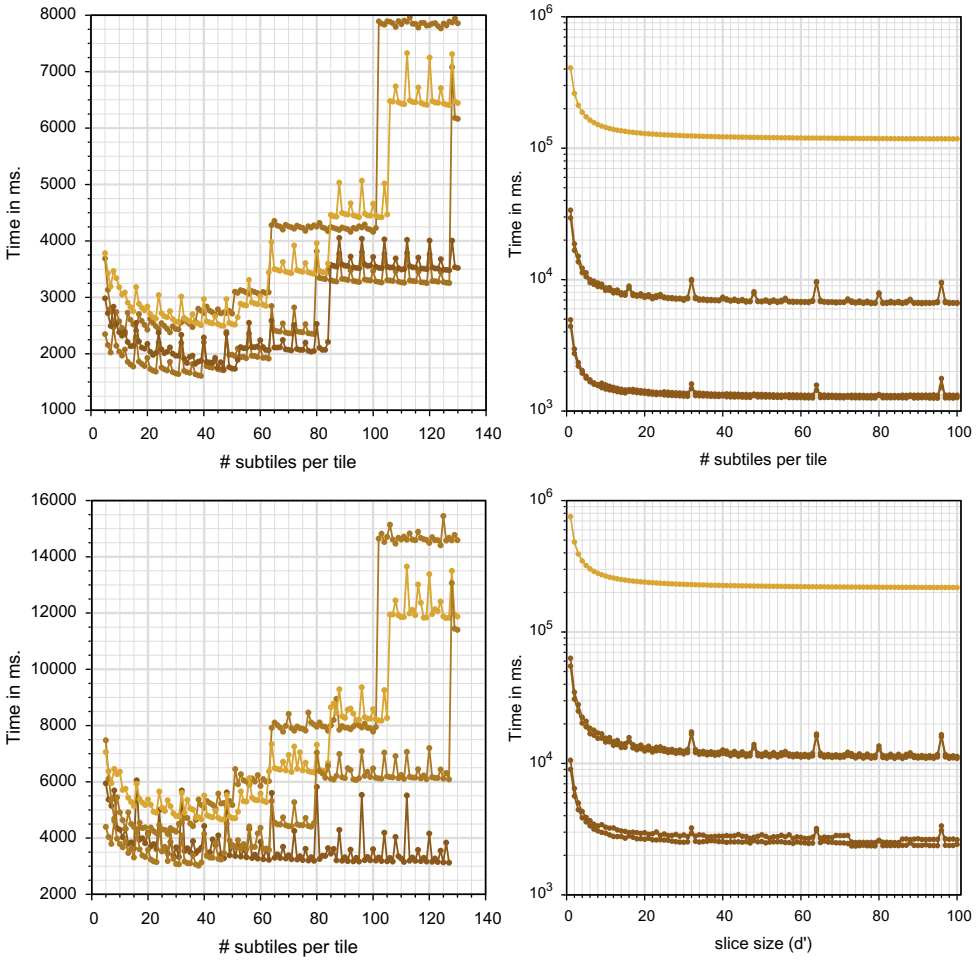

Fig. 9. Varying d' values. Input sizes are $n_1 = n_2 = 2000, d = 5419$ (top plots), and $n_1 = n_2 = 1000$, d = 40,000 (bottom plots), on Nvidia FX5800 (left plots) and M2090 (right plots) GPUs. Lighter shades represent smaller block sizes.

ics processor, best performance is obtained when the block sizes are chosen to be c = 16, and r = 8, the number of subtiles s is in the range of 3 to7, and slice dimension d' is about ~50. While, on NVIDIA M2090, we see best performances for the values $c = 16, r = 8, s \sim 20$ and $d' \sim 50$. Note that the order of the choice of these parameters does not matter because each set of values represents a unique point in the parameter search space. Using the above analysis and reasonings, in the following we derive some useful conclusions on how to choose the values of the parameters to ensure high performance for the generalized all-pairs computation given any other GPU architecture without the need to experiment with every possible combination from the search space. A large value of c is beneficial given that there is still enough concurrency in the computations, and the shared memory and register resources are sufficient. In general, both c and r should be large, preferably with $c > r$. This enables more reuse of the input column vectors loaded into the shared memory. Also, c should preferably be a multiple of 16 (half warp-size) to ensure saturated and coalesced memory accesses. If a GPU architecture provides a large number of SMs, such as our FX5800, more concurrency is needed which would require decreasing c and r for a given problem size. On the other hand, if the problem size is increased, for a given number of SMs, enough concurrency is ensured, allowing increase of the values of c and r, as we saw for M2090 architecture. Furthermore, with larger shared memory and register resources, increasing the block size will benefit from it.

Optimal values for c and r are the most essential to obtain high performance, which can improve the performance by over an order of magnitude. Once a block size is chosen, further performance improvement is achieved by a balanced choice of the other two parameters: s and d'. Again, to ensure enough concurrency, s should be kept small when applicable based on the architecture. If the problem size is increased, or number of SMs is decreased, increasing s would improve performance due to increased data reuse. As a general rule of thumb, s should be kept on the lower side as long as it is greater than 1. The choice of s is independent of shared memory and register resources available.

Given ample shared memory and registers per SM, the degree of concurrency is not affected by d'. Also, as seen from the same behavior in the graphs in Fig. 9, changing input size (d and n) does not affect the choice of d'. For a small shared mem-

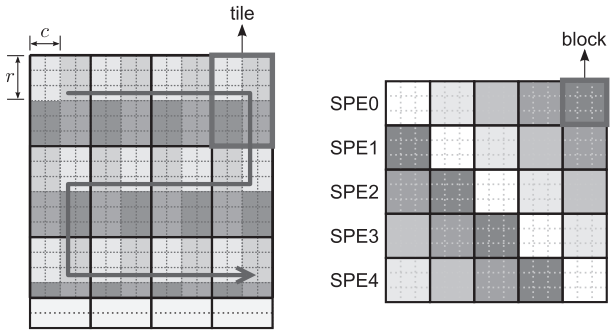

Fig. 10. A decomposition scheme on the Cell processor. D is decomposed into tiles (left), and each tile is decomposed into blocks (right) of size $r \times c$. A snake-like space filling curve is used to move computations from one tile to the next as shown by the arrow. Blocks shown in the same shade of gray are computed simultaneously in an iteration by the corresponding SPEs marked.

ory size, a smaller d' ensures enough warps can be scheduled on an SM. Hence, if the shared memory size is increased, increasing d' would deliver better performance.

5. An efficient scheme for the Cell processor

The IBM Cell processor [19,20] is a heterogeneous multi-core CPU, offering high-performance through specialized vector processing cores (SPEs). In previous work, we developed an optimal scheme for scheduling all-pairs computations on the Cell processor. Here we give a brief high-level overview of the scheme for the purpose of comparing with the GPU parallelization and to help understand the performance comparisons provided subsequently. Further details can be found in [21,11,5].

5.1. Decomposition into tiles, blocks and slices

The idea for performing the desired computations is the similar on this platform as on GPUs, but differs in details of the lower level decompositions because the Cell provides a coarse-grained parallelism (16 SPEs on one Cell blade), while GPUs provide fine grained parallelism (240 to 512 CUDA cores on the GPU used in our experiments). Also, unlike GPUs, each SPE core can directly communicate and transfer data to another SPE. We exploit these features in our scheme on the Cell processor.

We decompose the computations of matrix D into *tiles*, such that a single tile can be computed simultaneously in parallel among all the SPE cores on the Cell processor. A tile is further decomposed into *blocks*, where input and output for a single block can be stored in the limited Local Store (LS) of an SPE. This decomposition is shown in Fig. 10.

Computation of a tile is done in p stages, assuming p SPEs. Initially each SPE loads input vectors from the main memory for its corresponding first block. Then, in each iteration an SPE computes one block, and transfers the corresponding column vectors to the next SPE before moving onto compute the next block. Tiles are computed one after the other. The number of memory transfers is minimized by an optimal choice of the block size, computed analytically, as well as by following a snake-like space filling curve when moving from one tile to the next. These ensure the maximum possible reuse of the input vectors loaded into the LS of each of the SPEs. For higher dimensions, we follow the same slicing technique for the input vectors as described earlier for GPUs. We use this implementation on the Cell processor, and contrast it with the GPU implementation performance in the following.

6. Performance analysis

In this section, we present performance results of the proposed GPU-based all pairs computation method for different problems sizes, dimensionality, and precision. We use the parameters $c = 16$, $r = 6$, $s = 4$ and d'=50, which provide an optimal choice based on experiments conducted. We further provide comparison of GPU performance with the Cell using the scheme described in Section 5, and multi-core implementations using OpenMP and Intel Threading Building Blocks (TBB). For the latter comparison, we implemented a multi-core parallelization of the all-pairs computation using OpenMP (OMP) [22] with C++ to parallelize the outermost loop in the computation of D, which iterates over all the input vectors from the matrix M_1. In the second multi-core implementation, we employed Intel TBB [23,24]. Here, we used the two-dimensional iteration space class `tbb::blocked_range2d` to define the range of computations in output matrix D, which is used by the `tbb::parallel_for` routine to parallelize the computations. In all the four implementations, we use the L_p-norm distance metric as the computational kernel.

We present performance results on the following four platforms:

1. **GPU 1**: NVIDIA Quadro FX 5800 (GT200GL), 4 GB DDR3 device memory, CUDA 3.0. This has 30 multiprocessors, each with 8 CUDA cores, totaling 240 CUDA cores, and 16 KB shared memory.
2. **GPU 2**: NVIDIA Tesla M2090, 5 GB DDR3 device memory, CUDA 4.2. This has 16 multiprocessors, each with 32 CUDA cores, totaling 512 CUDA cores, 48 KB shared memory and 16 KB L1 cache.
3. **Cell**: IBM QS22 Cell blade, equipped with dual PowerXCell 8i 3.2 GHz processors, 4 GB DDR2 main memory, with Cell SDK 3.1. Once such blade provides a total of 16 SPE cores, each with 256 KB Local Store.
4. **CPU**: Intel Nehalem based dual-socket quadcore Xeon (E5504) 2 GHz processors, 12 GB DDR3 main memory. This provides a total of 8 cores. L3 cache is 8 MB shared among four cores on a chip, L2 cache is 256 KB percore, and L1 cache is 32 KB per core. We use this platform to execute both OpenMP and Intel TBB implementations.

6.1. Single precision performance results

The performance results of the four implementations with single precision L_p-norm kernel, on data sets with varying number of input vectors $n_1 = n_2 = n$ is shown in Table 1 for $d = 5419$ and in Table 2 for $d = 40{,}000$.

In Fig. 11 we show the corresponding speedups obtained by each of the four parallelized implementations with respect to a single-precision sequential implementation executing on a single core of the Intel Xeon processor.

Our GPU and Cell implementations outperform the straight-forward CPU parallelizations with OpenMP and Intel TBB. Furthermore, the GPU implementation outperforms the Cell implementation by a factor of 16 to 18. One of the contributing factors for this is that the GPU provides a massive fine-grained parallelism, which is well suited for the type of computations at hand. Another contributing factor is the peak main memory bandwidth, which is 25.6 GB/s on the Cell and 102 GB/s on the FX5800 GPU. The effect of this can be seen when we compare the speedups of GPU implementation over the Cell implementation for the two values of d. In the case when $d = 40{,}000$, we obtain a gain of around 18, while it is around 16 for $d = 5419$.

Table 1
Execution times in seconds for single precision L_p-norm computations with $d = 5419$.

$n_1 = n_2$	Cell	FX5800	M2090	Xeon-OMP	Xeon-TBB
1000	7.21	0.42	0.35	11.60	19.22
2000	27.19	1.62	1.28	46.11	75.20
3000	61.75	3.69	2.87	104.24	172.54
4000	107.92	6.65	5.09	184.59	336.44
5000	170.85	10.49	7.95	288.53	472.59
6000	243.62	15.25	11.41	415.39	693.75

Table 2
Execution times in seconds for single precision L_p-norm computations with $d = 40{,}000$.

$n_1 = n_2$	Cell	FX5800	M2090	Xeon-OMP	Xeon-TBB
1000	54.74	3.02	2.55	84.32	139.08
2000	216.63	11.95	9.41	337.28	581.28
3000	484.27	27.16	21.14	756.67	1245.63
4000	864.22	48.62	37.46	1344.92	2216.40
5000	1356.70	76.33	58.62	2101.17	3514.31
6000	1949.71	111.11	84.08	3027.57	5006.19

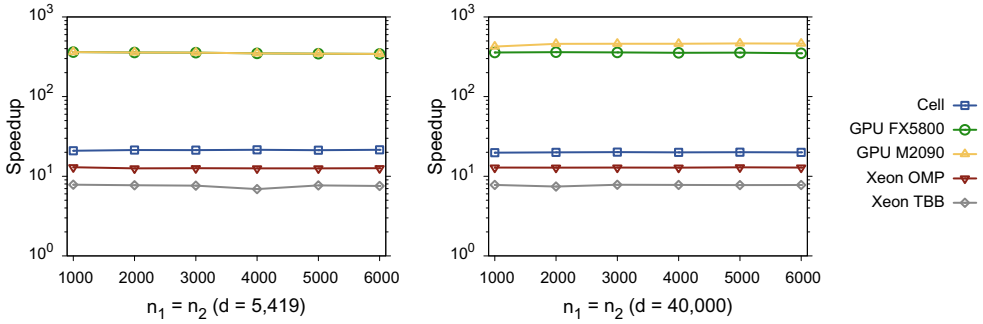

Fig. 11. Single precision performance speedups for STI Cell, FX5800 GPU, M2090 GPU, OpenMP on Xeon CPU and TBB on Xeon CPU implementations, with respect to a sequential implementation running on a single core of Intel Xeon processor.

Compared to a sequential implementation executing on a single core of Intel Xeon processor, GPU implementation achieves a speedup of around 350, while Cell implementation achieves a speedup of 20, as seen in Fig. 11. We also observe that we obtain super-linear speedups for our parallel CPU implementations. This is quite a common phenomenon and is attributed to the multi-level caches available on the CPU. Due to the presence of this hierarchy of caches, reuse of data greatly increases among all the cores (*cache-effect*).

We also note that OpenMP parallelization outperforms the Intel TBB parallelization for these single precision results. This is expected because OpenMP works best with parallelizing large and predictable data parallel problems with independent computations in for-loops, which is the case with generalized all-pairs computations. Intel TBB is best suited for parallelizing problems with less structured or less consistent parallelism, and TBB would outperform OpenMP. Furthermore, Intel TBB takes an object-oriented path for defining parallelism. It is generally more scalable for a large number of computations because it defines parallelism in terms of tasks and not threads, as is the case with OpenMP. The task creation and scheduling mechanisms in TBB also contribute to some overhead than that by thread creation and scheduling in OpenMP which is performed by the compiler ahead of the executions. Hence the choice of the compiler in the case of OpenMP is crucial. We used the Intel 11.1 version compilers for our tests. Intel TBB does not need any special compiler support. Other researchers have also compared the performance of scientific applications with OpenMP parallelization versus Intel TBB, for example see [25], which support our observations.

6.2. Double precision performance results

The performance results with double precision L_p-norm kernel on data sets with varying number of vectors $n_1 = n_2 = n$ is shown in Table 3 for $d = 5419$ and in Table 4 for $d = 40{,}000$. The corresponding speedups of the four implementations, when compared to a double-precision sequential implementation executing on a single core of the Intel Xeon processor are shown in Fig. 12.

Table 3
Execution times in seconds for double precision L_p-norm computations, with $d = 5419$.

$n_1 = n_2$	Cell	FX5800	M2090	Xeon-OMP	Xeon-TBB
1000	21.75	1.73	0.33	21.30	20.73
2000	86.19	6.89	1.27	84.98	87.47
3000	194.33	15.45	2.86	192.39	178.23
4000	344.41	27.43	5.04	339.67	319.99
5000	536.45	42.90	7.19	530.59	514.57
6000	770.58	62.12	11.35	764.17	722.40

Table 4
Execution times in seconds for Double Precision L_p-norm computations, with $d = 40{,}000$.

$n_1 = n_2$	Cell	FX5800	M2090	Xeon-OMP	Xeon-TBB
1000	162.76	12.73	2.40	156.62	147.57
2000	639.71	50.57	9.33	625.25	583.31
3000	1455.87	113.47	20.99	1405.45	1415.03
4000	2574.52	201.51	37.01	2497.58	2311.87
5000	4039.97	315.15	58.13	3902.14	3766.18
6000	5794.40	454.16	83.29	5620.04	5763.62

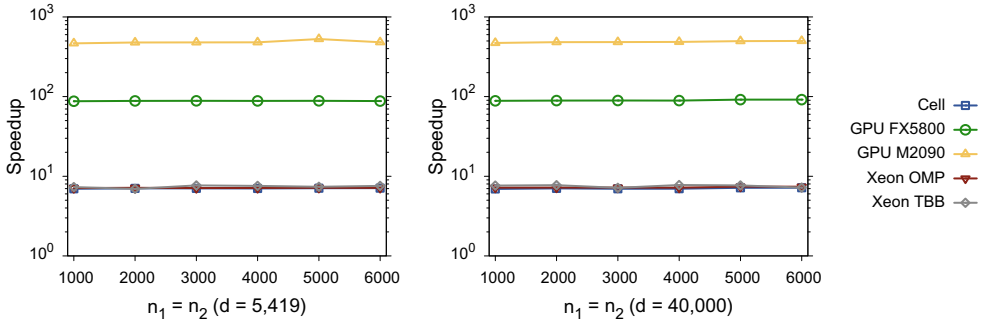

Fig. 12. Double precision performance speedups for STI Cell, FX5800 GPU, M2090 GPU, OpenMP on Xeon CPU and Intel TBB on Xeon CPU implementations, with respect to a sequential implementation running on a single core of Intel Xeon processor.

Similar to the single precision results, the GPU implementation outperforms both the CPU based parallelizations. An interesting observation here is that the factor of improvement the GPU implementation provides over the Cell implementation is around 12, which is less than that for single precision we saw earlier. This is because the support of double precision on the GPU architecture we used is not as good as single precision. Each SM on this GPU provides a single 64-bit FPU, making double precision performance much lower. Furthermore, the GPU implementation achieves a speedup of around 90 and Cell implementation around 7, when compared to a sequential execution on a single core of Intel Xeon processor. We also note that the performance of Cell implementation is similar to that of the other CPU implementations. With double precision, the memory requirements grow, and less number of vectors can be stored in the LS of an SPE on the Cell processor, requiring more number of memory transfers. Intel Xeon CPUs provide a full double precision support, and their large caches (8 Mb L3, 256 KB L2 per core, and 32 KB L1 per core) play to their benefit with such memory intensive computations.

7. Conclusions

In this paper we developed efficient and scalable architecture-aware techniques for the problem of scheduling generalized all-pairs computations on graphics processors. These techniques are based on decomposition of the output matrix into tiles, thread blocks, and subtiles, and decomposition of the input into dimensional slices. We focus on the most efficient usage of the available memory hierarchies and minimizing the number of memory transfers taking place. This is crucial for high-performance of such memory-intensive computations on these architectures. Our methods also make full use of the parallelism hierarchy available with the multi-level decompositions, and maximum data reuse at each level. Further, we investigated and analyzed the effects various values of these decomposition parameters have on the performance of the GPU implementation. Using the extensive experiments we conducted, we presented guidelines for the reader on how to choose these parameters to optimize performance on a given GPU architecture. To facilitate comparison with other emerging architectures, we provided detailed comparisons with our optimized implementation on the Cell platform and multi-core parallelizations using OpenMP and Intel Threading Building Blocks on Xeon CPUs, for both single precision and double precision, and show that specialized massively parallel architectures like GPUs are the key for the future of high-performance computing. Our library for scheduling the all-pairs computations on graphics processors is open source and is available under LGPL.

Acknowledgments

The authors thank Baskar Ganapathysubramanian for providing input data sets from flapping-wing MAV simulations and microstructure samples. The all-pairs computations work on the Cell processor was done previously in collaboration with Jaroslaw Zola. The authors also acknowledge Georgia Institute of Technology, its STI Center of Competence, and the National Science Foundation, for the use of Cell resources that have contributed to this research.

References

[1] B. Hendrickson, S. Plimpton, Parallel many-body simulations without all-to-all communication, Journal of Parallel and Distributed Computing 27 (1995) 15–25.
[2] J. Zola, M. Aluru, S. Aluru, Parallel information theory based construction of gene regulatory networks, in: Proceedings of the 15th annual IEEE International Conference on High Performance Computing HiPC'08, LNCS, vol. 5375, 2008, pp. 336–349.
[3] P. Berkhin, A survey of clustering data mining techniques, in: Grouping Multidimensional Data, Springer, 2006, pp. 25–71.
[4] M. Vikram, A. Baczewzki, B. Shanker, S. Aluru, Parallel accelerated cartesian expansions for particle dynamics simulations, in: Proceedings of the 24th IEEE International Parallel and Distributed Processing IPDPS'09, Symposium, 2009, pp. 1–11.
[5] J. Zola, A. Sarje, S. Aluru, Constructing gene regulatory networks on clusters of Cell processors, in: International Conference on Parallel Processing, 2009, pp. 108–115.
[6] J. Zola, M. Aluru, A. Sarje, S. Aluru, Parallel information-theory-based construction of genome-wide gene regulatory networks, IEEE Transactions on Parallel and Distributed Systems 21 (12) (2010) 1721–1733.
[7] N. Arora, A. Shringarpure, R. Vuduc, Direct N-body kernels for multicore platforms, in: Proceedings of International Conference on Parallel Processing ICPP'09, 2009, pp. 379–387.
[8] A. Wirawan, B. Schmidt, C.K. Kwoh, Pairwise distance matrix computation for multiple sequence alignment on the Cell Broadband Engine, in: Proceedings of the 9th International Conference on Computational Science ICCS'09, 2009, pp. 954–963.
[9] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, E.S. Quintana-Ortí, G. Quintana-Ortí, Exploiting the capabilities of modern GPUs for dense matrix computations, Concurrency and Computation: Practice and Experience 21 (18) (2009) 2457–2477.
[10] D. Chang, A.H. Desoky, M. Ouyang, E.C. Rouchka, Compute pairwise manhattan distance and pearson correlation coefficient of data points with GPU, in: International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2009.
[11] A. Sarje, J. Zola, S. Aluru, Accelerating pairwise computations on Cell processors, IEEE Transactions on Parallel and Distributed Systems 22 (1) (2011) 69–77.
[12] NVIDIA Corporation, NVIDIA Programming Guide 3.0, February 2010 (last accessed).
[13] B. Ganapathysubramanian, N. Zabaras, A non-linear dimension reduction methodology for generating data-driven stochastic input models, Journal of Computational Physics 227 (2008) 6612–6637.
[14] J.C. Caruso, N. Cliff, Empirical size, coverage, and power of confidence intervals for Spearman's rho, Educational and Psychological Measurement 57 (1997) 637–654.
[15] C.O. Daub, R. Steuer, J. Steuer, S. Selbig, Kloska, Estimating mutual information using B-spline functions – an improved similarity measure for analysing gene expression data, BMC Bioinformatics 5 (118) (2004).
[16] Y. Moon, B. Rajagopalan, U. Lall, Estimation of mutual information using kernel density estimators, Physical Review E 52 (3) (1995) 2318–2321.
[17] Z.J. Wang, Vortex shedding and frequency selection in flapping flight, Journal of Fluid Mechanics (2000) 323–341.
[18] NVIDIA Corporation, NVIDIAs Next Generation CUDA Compute Architecture: Fermi, White paper, 2009.

[19] T. Chen, R. Raghavan, J.N. Dale, E. Iwata, Cell Broadband Engine architecture and its first implementation: a performance view, IBM Journal of Research and Development 51 (5) (2007) 559–572.
[20] IBM Corporation, Cell Broadband Engine resource center, 2010. <http://www.ibm.com/developerworks/power/cell>.
[21] A. Sarje, Applications on emerging paradigms in parallel computing, Ph.D. thesis, Iowa State University, 2010.
[22] OpenMP.org, The OpenMP API specification for parallel programming, 2011. <http://openmp.org>.
[23] Intel Corporation, Intel Threading Building Blocks: Reference Manual, Intel Corporation, April 2010 (last accessed).
[24] J. Reinders, Intel Threading Building Blocks, first ed., O'Reilly Media, Inc., 2007.
[25] P. Kegel, M. Schellmann, S. Gorlatch, Using openmp vs. threading building blocks for medical imaging on multi-cores, in: Proceedings of the 15th International Euro-Par Conference on Parallel Processing Euro-Par'09, Springer-Verlag, 2009, pp. 654–665.