A High-Performance Multi-Element Processing Framework on GPUs

Linh Ha, James King, Zhisong Fu and Robert M. Kirby

UUSCI-2013-005

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

August 26, 2013

Abstract:

Many computational engineering problems ranging from finite element methods to image processing involve the batch processing on a large number of data items. While multielement processing has the potential to harness computational power of parallel systems, current techniques often concentrate on maximizing elemental performance. Frameworks that take this greedy optimization approach often fail to extract the maximum processing power of the system for multi-element processing problems. By ultilizing the knowledge that the same operation will be accomplished on a large number of items, we can organize the computation to maximize the computational throughput available in parallel streaming hardware. In this paper, we analyzed weaknesses of existing methods and we proposed efficient parallel programming patterns implemented in a high performance multi-element processing framework to harness the processing power of GPUs. Our approach is capable of levering out the performance curve even on the range of small element size.

A High-Performance Multi-Element Processing Framework on GPUs

Linh Ha, James King, Zhisong Fu and Robert M. Kirby
Scientific Computing and Imaging Institute
University of Utah
Email: {lha, jsking, zhisong, kirby}@sci.utah.edu

Abstract—**Many computational engineering problems ranging from finite element methods to image processing involve the batch processing on a large number of data items. While multi-element processing has the potential to harness computational power of parallel systems, current techniques often concentrate on maximizing elemental performance. Frameworks that take this greedy optimization approach often fail to extract the maximum processing power of the system for multi-element processing problems. By ultilizing the knowledge that the same operation will be accomplished on a large number of items, we can organize the computation to maximize the computational throughput available in parallel streaming hardware. In this paper, we analyzed weaknesses of existing methods and we proposed efficient parallel programming patterns implemented in a high performance multi-element processing framework to harness the processing power of GPUs. Our approach is capable of levering out the performance curve even on the range of small element size.**

I. Introduction

A multi-element processing operation consists of three main ingredients:

1) **Fundamental action** - A core operation acting on some logical unit of data (*an element*).
2) **Data** - A large number of elements on which the action is applied.
3) **Constraints** - The action has elemental independence. The result neither depends on the order the data is traversed, nor requires partial computation of the data to be completed.

These ingredients often exists when we perform operations on multiple element structures such as 2D/3D unstructured meshes or n-D volume data in scientific computing and imaging as well as multi-record data mining and information exploration. These operations include but not limit to local computations such as computation of tangent vector field of a 3D surface mesh or sliding window computations such as the image convolution. In scientific computing, these operations are often referred through the represented action in the batch processing mode, we however present the multi-element terminology to focus on the uniformity of the action and data which allows us efficient parallel implementations.

Multi-element processing can be seen in many scientific and industrial applications, for example in traditional and high-order finite element analysis [28], [31], [42], [47], robust image processing [2], [11], [26], [46], [50], photo tourism [22], [44], video processing [10], [21], [40], super-resolution [6], [9], and population analysis [18], [24], [39] to name a few.

The total amount of computation and memory consumption of a multi-element operation are proportional to the complexity of the action as well as the number of elements and are often so high that parallel processing is required for real-time processing. While this brings the potential to harness the computational power of modern multi-processor and SIMD systems such as graphical processing units (GPUs), current processing frameworks [1], [49] fail to fully exploit the processing power of these processing devices for multi-element algorithms [33]. The main reason is that existing frameworks often concentrate on maximizing the performance of an operation on a single large size element.

More specifically, the large computational requirements of multi-element processing often come from a large number of input data elements rather than the size of each element. Often, the computation and memory transfer of a single element is so small that a single element processing techniques will not be able to saturate the computation and memory bandwidth of a parallel processing system. This inefficiency is then multiply by a large number of elements resulting in wasted computational resources and inadequate performance.

A potential solution is to use concurrent execution mechanisms [17] which allow multiple devices to execute multiple functions simultaneously when the computational resources are available. However, this approach involve constraints and has very limited influence in practice [43]. Beside, it is highly-driven by advanced supporting features of the platform, leading to a platform dependent and unscalable solution.

Another reason for poor performance with multi-element processing is the implicit synchronization at the boundary of function calls [14], [17]. Due to arbitrary complexity and dependency of high level functions, this synchronization is required to maintain the correctness of a parallel algorithm. In particular, in a general for loop, there's no guarantee that the next iteration does not consume the result of the current iteration. The synchronization adds overheads because the stages of the coprocessor device need to be reset and all the processing IOs need to be finalized. The wide range of the input parameters of a multi-element processing add the difficulty. While the overhead is negligible in high range when the element size is large, it is significant in comparison with the actual computations required by the algorithms when the element size is small. Even worse, the overhead is multiplied by the size of the for loop or the number of elements.

To reduce call overhead, advanced parallel processing systems [14], [17] offer asynchronous processing mechanisms [36] which allow developers to overlap calls of multiple asynchronous streams. Consequently, the systems enable a seamless transformation from a single processing algorithm to its multi-processing counterpart. However, only the runtime cost is hidden, the overhead is still exist. In other word, computational resources are still consumed to process this redundancy, and hence the overall efficiency is lowered.

We have searched for the answer for the performance problems using fine-grain parallelism optimization and loop unrolling techniques.Our experiment shows that not a single parallel strategy and data structure can achieve satisfied performance over a wide range of input parameters. The optimal solution however requires multiple strategies and data structures, each need to be applied on certain range of the number of elements as well as the problem size. This requires a multi-element processing system to provide both optimization strategies and algorithm selection mechanism. However, the latter is the subject of auto-tuning and hence is not the focus of the paper. Here we concentrate on the former by providing the abstract multi-element parallel programming patterns which can be realized to optimal solutions using compiler guidance techniques or template meta-programming. We aim to a high level of abstraction in our presentation to be able to cover a large range of multi-element processing problems. The optimization of a certain problem is problem dependent that can be achieved through a specialization process available with template meta-programming.

In this paper, we introduce a novel software solution based on multi-element concept. Here, we exploit *a priori* that the same operation is to be accomplished on a large number of items (1)to organize the computation to maximize the use of available parallel computing resources and (2) to unify the actions in a single on device function to remove the overhead entirely. Our framework exploit fine-grain parallel optimization to handle a range of element sizes from the very small to the moderately large. Consequently, our method is able to leverage the performance curve over entire range of input.

To demonstrate the efficacy of our approach, we have developed a GPU APIs for batch processing of linear algebra operations such as matrix-vector multiplication, matrix-matrix multiplication and linear system solving (*e.g.* Cholesky decomposition with backsolve). There are two main reasons for choosing linear algebra operations as the illustration for our solution efficiency. First, linear algebra operations cover a wide range of complexity from simple linear with BLAS level one functions to high polynomial power complexity with BLAS level 3 and LAPACK functions, so it is a good measurement for the generality of our framework. Second, linear algebra packages have been the most widely-used components in scientific computing applications and its efficiency is critical to many real-time applications.

We present the results of using our system for a myriad of matrix ranks, from the very small to the moderately large – a range often neglected by many competing systems. We show that by exploiting the additional information added when performing multi-element processing, one can utilize the available streaming massively SIMD hardware most efficiently. In particular, we compare our result with NVIDIA CUDA Basic Linear Algebra Subroutines (cuBLAS) APIs, which has been longtime developed, highly optimized and constantly improved over time. Our results show that our method has speed up many functions significantly from one to two magnitude orders on small element range.

The paper is organized as follows. In Section II, we outline the relevant previous work related to multi-element processing. In Section III, we present the algorithmic overview of our multi-element processing framework, and describe the implementation details of our effort. In Section IV, we present results demonstrating the efficacy of our framework when run on currently available streaming massively SIMD hardware. We then present our conclusions and vision for future work in Section V.

II. Related Work

Our research was primarily motivated by an interest in seeing how one might efficiently use the GPU when solving the Poisson equation with the Hybridized Discontinuous Galerkin (HDG) Method [32]. In that problem setting, we needs to generate and operate on a large number of tiny to small sized matrices in order to form the local systems used within the HDG formulation. In our attempt to formulate an abstraction of this problem, we realized that this same multi-element processing idea can be found in a multitude of application domains (as aforementioned). The design of such a system also transcends a large number of discipline-areas within computer science: numerical methods, parallel languages, GPU computing, performance tuning, compiler optimization and parallel optimizations. In this section, we cannot cover all the previous work that might overlap in some way with our topic focus. Instead, we outline here some of the major works we considered when formulating and implementing our multi-element framework.

A. CPU-related previous work

Multi-element processing is a common technique employed in both traditional and high-order finite element methods (FEM) [28], [31]. The huge memory bandwidth and computational requirements of these problems naturally demand parallel strategies. Our work is motivated from FEM research by Kirby *et al.* [32], [33] which concentrated on analyzing the complexity and stability of different spectral element methods (Continuous Gelerkin CG, Discontinuous Galerkin DG, and HDG) to solve computational fluid dynamics problems. These techniques require local computation on discrete elements, a typical example of multi-processing functions. An implementation of these methods often excessively use the highly-optimized linear algebra packages in basic level (BLAS functions) such as Atlas, Goto Blas, Intel MKL, AMD ACML [29], [30], [51] and high-level (LAPACK functions) such as LINPACK, ScaLAPCK, PLAPACK [7], [48]. As far as we are aware of, these packages are specifically designed to provide optimal solutions for single element processing; the batch processing which can be used for multi-element problems is available in some cases but with limited support.

One reason for this lack of support partially comes from potential solutions using compiler optimization or parallel language support. OpenMP [37], a compiler directive approach, harnesses the uniform for-loop like nature of multi-element processing to map computations to multi-threaded CPU architectures. In fact, this is a favorable solution on CPUs as it harnesses the compiler optimization techniques which have long been in research and grown to a mature level. The method also provides portability and maintainability. However, the complexity of heterogeneous processing systems (GPUs, Cells, FPGAs) is higher versus homogeneous CPU-based systems, which makes hardware execution mapping a challenging task and consequently compiler optimization a less effective approach. We exploit the unified architecture of GPU systems to guide the mapping process, to ease compiler duty and hence to make compiler optimization more efficient.

Parallel programming languages such as MPI, NESL, HPF *etc.* [8], [23] provide solutions for a large range of problems including both single and multi-element processing. The techniques have been successfully applied on supercomputing for large processing systems such as IBM SP-2, CM5, Cray C90 and J90, MasPar MP2 *etc.* However, as they are designed for large machines to solve ultra-large problems, these languages are not optimized for problems in smaller-scaled architectures. Also these languages have limited support on heterogeneous processing systems and are incapable of harnessing available processing power from these systems.

B. GPU-related previous work

In the last decade, GPUs have arisen as a driving platform for heterogeneous parallel processing with strong scalability, power and computational efficiency [12]. In the past few years, a number of algorithms have been developed to harness the processing power of GPUs for a number of problems which require multi-element processing techniques [5], [38], [43]. However, as far as we know, there is not a single publication to address the multi-element processing problem in general on GPUs. Our paper is the first to generalize the existing solutions from published works. We also combine the techniques derived from single element processing including kernel configuration and auto-tuning, high efficiency memory access, and asynchronous streaming.

Finding efficient implementations for solving linear algebra problems is one of the most active areas of research in GPU computing. The NVIDIA CuBlas [17] and AMD APPML [15] are well-known solutions for BLAS functions on GPUs. While CuBlas is specifically designed for the NVIDA GPU architecture based on CUDA [17], the AMD solution using OpenCL [3] is a more general cross platform solution for both GPU and multi-CPU architectures. CuBlas has constantly improved based on a successive number of research attempts by Volkov *et al.* [49], Dongarra *et al.* [1], [45] *etc.* This results in a improvement of one or two orders of magnitude speed-up for many functions from the first release version till now. On the high level, MAGMA developed by ICL team provides optimized LAPACK functions for both NVIDIA and AMD hardware [1]. In recent releases, CuBlas and MAGMA have been providing batch processing support to improve processing efficiency on multi-element processing tasks. The support is, however, neither complete nor efficient and scalable due to the constraints of their approaches. This motivates us to find of a more effective and scalable approach and also a more general solution rather than a linear algebra package for multi-element processing.

Segmented operations and data structures derived from prefix sum computation [13] provide solutions for many non-uniform multi-element processing problems [4]. The CUDPP [25] and Thrust library [27] provide highly efficient parallel implementations of scan-based operations used as basic algorithm blocks for high performance GPU sorting, multi-grid computation, data compression, LU decomposition *etc.* Though segmented scan algorithms and data structures can handle a number of uniform multi-element processing problems, as an indirect approach it does not provide a general solution. It also fails to exploit the regularity of the operations and data structures leading to more memory bandwidth and storage requirements, and as a result is less efficient. We overcome their limitation using uniform data structures and algorithms, meanwhile we still exploit their optimization techniques for scan-based functions.

One important aspect of our paper focuses on optimization techniques to apply with multi-element processing. We make use of existing optimizations on global memory such as memory caching [36], pointer redirecting [35], explicit data blocking [34], and data padding [36]; however in the context of the multi-element processing problem we provide new insight to these techniques. We also exploit the results of performance tuning techniques from CuBlas [49], MAGMA [20], [34] and Thrust [27] to propose our adaptive selection strategies and automatic kernel configuration based on occupancy calculation [16].

```
1: Input : data pointers, $n$ element size, $nE$ No. of elements
2: Output: multi-element operation
3: #pragma omp parallel for                ▷ OpenMP directives
4: for $i \leftarrow 1, nE$ do
5:     Update data pointer to the $i^{th}$ element
6:     $SPA\_kernel <<< SE\_cfg, NULL >>>$(...)          ▷
   Single element call with configuration $SE\_cfg$
7: end for
```

Algorithm 1: Regular for-loop MPA

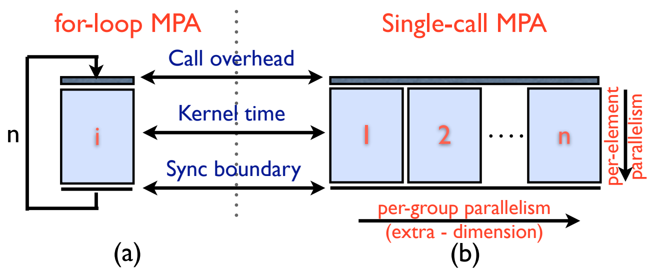


Fig. 1. Single call MPA solution effectively removes the redundancy of a for-loop MPA approach and adds an extra dimension of parallelism to allow for scalability.

III. Multi-element processing framework

A multi-element processing algorithm (MPA) often exploits independency between elements, hence it can be considered a multi-session processing pipeline. This makes a multi-element algorithm a trivial extension of the single element algorithm (SPA) using a for-loop over all elements (Figure 1(a) and Algorithm 1). However, this simple approach adds overhead to the processing time, resulting in inadequate performance of an MPA. The performance drop due to added overhead is significant when the (data) element sizes are small. Unfortunately many applications employ elements with small size [6], [19], [31], [32], [46], [47].

Another problem with a for-loop approach is scalability. A simple for-loop is inherently sequential, hence the scalability of the algorithm depends on the scalability of the single element algorithm. However, parallel algorithms often have poor scalability when involving small data element sizes.

Even though the sequential execution problem could be addressed, in theory, using a parallel stream execution [36], batch processing mode [43] (see Algorithm 2) or implicit compiler directives parallelism such as OpenMP [37], these approaches do not remove the exiting redundancy. Moreover, their effectiveness and scalability are largely dependent on the different means of support provided by the systems used.

In our processing framework, we consider a multi-element algorithm as a problem by itself rather than an indirect generalization of the single element processing counterpart. We analyze the processing redundancy problem and manage to remove this overhead entirely when working on small data elements. To address the scalability problem, we express the parallelism in one dimension higher (see Figure 1(b)): in terms of both problem size and the number of elements. This will guarantee the scalability of our approach even when the element size is small. Note that our solution is not intended to be a replacement of general approaches with large element processing but a cooperative, adaptive range approach that allows for the selection of the best solution based on the problem size and available resources of the systems.

```
1: **Input** : data pointers, $n$ element size, $nE$ No. of elements
2: **Output**: multi-element operation
3: Create a stream array $st$ with $nE$ streams
4: **for** $i \leftarrow 1, nE$ **do**
5:     Update data pointer to the $i^{th}$ element
6:     $SPA\_kernel <<< SE\_cfg, st[i] >>>(...)$     ▷ Execute SPA on $i^{th}$ stream
7: **end for**
```
Algorithm 2: Batch mode MPAs using streams

A. Multi-element redundancy

A multi-element algorithm is often rather homogeneous: all elements can be assumed to have similar sizes and common parameters. If we consider an MPA as a parallel multi-section extension of a SPA using OpenMP [37] or a asynchronous processing methods [36], [43], the common parameters can no longer be shared but need to be duplicated. This copy is a source of redundancy (see Figure 1(a)).

To guarantee the correctness of a parallel algorithm, the memory and instruction pipeline are required to be persistent between function calls. As a result, a function is executed only when the system is in a resident stage: the instruction pipeline is reloaded, the IOs are completed and the executions are synchronized at the call boundary [17]. However, this is another redundancy since the processing between elements is often independent from the others. Persistence is only required when the entire operation finishes with all the elements (see Figure 1(b)).

We address the redundancy problem by implementing an MPA using a single GPU function. As we consider an MPA a single problem, we exploit the aforementioned uniformity to eliminate the unnecessary parameters copies and call overheads. We also remove all constraints due to persistent requirements. We employed existing GPU mechanisms to express the parallelism of MPA in order to manipulate and share common parameters.

We generalize GPU paradigms into two logical models that express the parallelism of MPAs: the thread-based execution model (*e.g.* Algorithm 3) and the block-based execution model (*e.g.* Algorithm 4). In the next section, we provide a discussion and comparison of these two logical models.

B. Thread-based execution model versus block-based execution model

The thread-based execution model (Figure 2(a)) implements each GPU thread as an execution unit for single element computation. Here we exploit instructional parallelism on the element level and GPU threading on the multi-element level. Because the number of threads equals the number of elements, the thread-based execution model scales well with the number of elements. However, as the resources per thread (*e.g.* number of registers, amount of shared memory) are typically small on most current GPUs, this strategy is only suitable to “tiny” problem sizes or with algorithms that are relatively memoryless (such as per thread reduction) .

The block-based execution model (Figure 2(b)) employs each execution block as the execution unit. Thus, the number

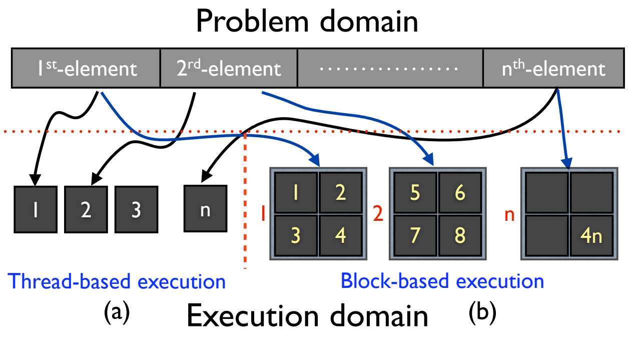


Fig. 2. Parallel execution mapping for multi-element processing based on the thread-based execution and block-based execution models.

of execution blocks equals the number of elements. We exploit the inner threading parallelism of each streaming multiprocessor (SM) on the element level and multi-SM parallelism on the multi-element level. All threads of a single execution block work together to complete tasks of a single element. Barriers are normally required to synchronize between a block's threads while a scratch memory space (*i.e* the GPU shared memory) is used to collaborate results. The granularity of the execution blocks controls the amount of resources available for each element's computation. As the resources are allocated per block, this strategy can handle a wide range of the inputs.

```
1:  Input : SoA input data, $n$ element size, $nE$ No. of
    elements, $bl$ block size
2:  Output: multi-element operation
3:  Call $MPA\_kernel <<< nE/bl, bl, NULL >>>$(...)
4:  function MPA_KERNEL(SoA data)
5:      __shared__ $s[blockSize][n]$
6:      $tid \leftarrow get\_threadId()$;
7:      $id \leftarrow get\_globalId()$;
8:      Load data from global-$id$ memory to $s[tid]$
9:      Execute $SPA$ on shared memory $s[tid]$
10:     Output $SoA$ data
11: end function
```

Algorithm 3: Thread-based MPA

```
1:  Input : input data in AoS format, $n$ element size, $nE$ No.
    of elements, $bl$ block size
2:  Output: multi-element operation
3:  Call $MPA\_kernel <<< nE, bl, NULL >>>$(...)
4:  function MPA_KERNEL(AoS data)
5:      __shared__ $s[n]$;
6:      $bid \leftarrow get\_blockId()$;
7:      Cooperative load data of $bid^{th}$ element to $s$
8:      __syncthreads__();          ▷ sync loading of $b^{th}$ block
9:      Coperative parallel $SPA$ on shared memory $s$
10:     __syncthreads__();▷ sync computation of $bid^{th}$ block
11:     Output $AoS$ data from shared memory to $bid^{th}$ output
12: end function
```

Algorithm 4: Block-based MPA

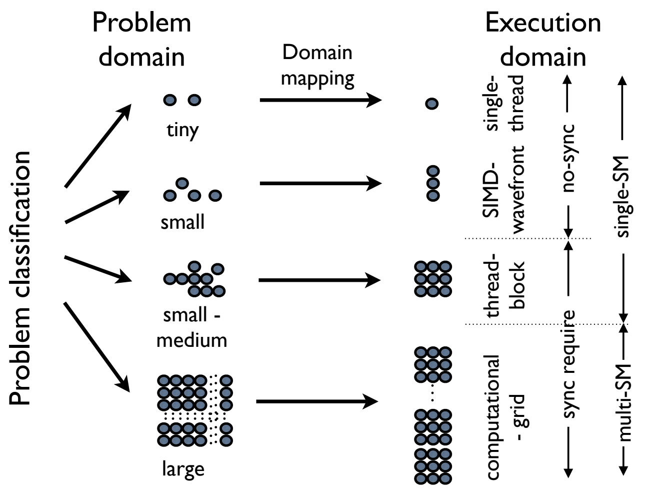


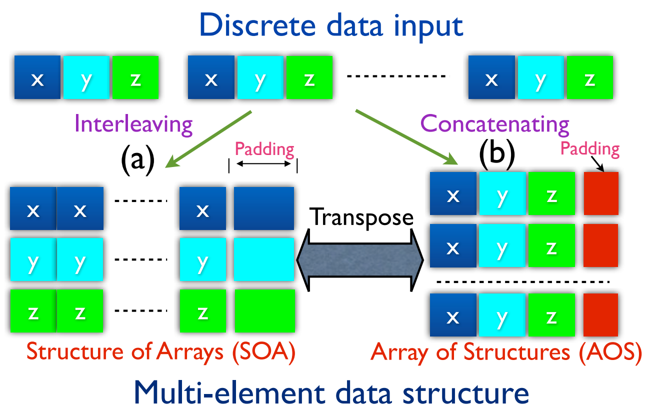
Fig. 3. Problem classification for algorithm mapping

In modern GPUs, the block-based execution models can be divided into wavefront (or SIMD-width), block (per-SM), and grid (using entire GPU) executions (illustrated in Figure 3). The wavefront is a very special case that requires no synchronization as all member threads execute in lock step. We have exploited this model to remove synchronization redundancy in reduction functions when the element size, for instance the vector size in this case, is less than 64. On the other end, we used the grid approach for large element size (above a hundred thousand) in conjunction with an asynchronous processing strategy [24]. This is different from the batch processing model from MAGMA [1] as our model is based on the optimal strategy initially proposed for multi-image processing tasks by Ha *et al.* [24]. In the current work we concentrate on the second of the block-based execution models presented, which maps all instructions per element to an SM, and hence we can exploit the shared memory resident on that SM. This is the most reasonable solution for the case when the element size is in the small to medium range (*i.e.* able to fit within shared memory), which is ineffectively covered by existing methods.

As the number of threads per SM and the number of SMs per GPU increases in modern systems, our thread-based and block-based execution models will scale well. Furthermore, by classifying the element size into different ranges and then applying appropriate execution models, we can amortize the performance of MPAs. Next, we discuss the data structures and algorithm mappings needed for efficient MPAs on GPUs.

C. Multi-element data structures

Data structures are crucial for determining performance. We have built a multi-element data structure from that of the single element processing data structure with the target of achieving maximum throughput. We evaluate the efficiency based on the coalesced condition [36]. Fortunately, this condition solely depends on the access pattern of neighboring threads, encouraging neighbor threads to access continuous data. For



Discrete data input
x y z
Interleaving
Concatenating
(a)
(b)
Padding
Transpose
Structure of Arrays (SOA)
Array of Structures (AOS)
Multi-element data structure

Fig. 4. Optimal data structure for multi-element processing (a) for the thread-based execution model and (b) for the block-based execution model.

the thread-based execution model, this turns out to be the interleaving data structure (see Figure 4(a)) where the first component of the data of the first element is laid out in the memory next to the first component of the second element and so on. For the block-based execution model, it becomes the concatenated data structure (see (Figure 4(b)) that lays out the data structure of each of the elements sequentially in memory.

The interleaving and concatenating data structures are well-known as the Structure of Array (SoA) and Array of Structure (AoS) respectively. Though SoA is generally preferred for single element GPU algorithms, in our multi-element processing framework it is employed only for the thread-based processing strategy or tiny-element problems while we exploit the AoS for a much wider data input range. Here we reaffirm our theory that there is no ultimate data structure that will serve all needs optimally, but rather we need to adapt the data structure to the algorithm strategy to achieve maximal performance.

1) High-performance data adapter: In our processing framework, we implement a problem-adapted approach for both algorithms and data structures. While the strategy provides maximal throughput per GPU kernel, it might introduce data mismatch between successive processing steps in the processing pipeline. Fortunately, we have an effective solution for this problem based on the data adapter modules: the transpose. As displayed in Figure 4, when we see the data structure in 2D space, the SoA turns out to be a transpose of the AoS; hence we can employ the optimal transpose implementation [41] (which is equivalent to a memory copy) to build this high-performance data adapter. We also combine the transpose with other transformed functions to employ different data structures for inputs and outputs, and hence we can save bandwidth if further transformations are needed (for example to take the square of each component value at the output).

2) Data padding strategy: To achieve the highest bandwidth efficiency, we apply data padding to guarantee the access of a thread/block starting at an aligned memory address. In particular, we employ data alignment per element with the AoS and data alignment per array with the SoA (see Figure 4). Our strategy has proven to be simple yet effective and compact. This alignment strategy only increases the storage by approximately 10 percent in contrast to the alignment per data dimension strategy which can be very expensive with high dimensional input data.

An additional benefit of data padding is that even though data might have odd size numbers in terms of number of elements and/or element size, our transpose function always achieves the highest memory bandwidth efficiency because execution blocks always access aligned memory for both data loading and storing. Hence, the added overhead due to the changing of data structures is minimized.

D. Multi-element algorithms

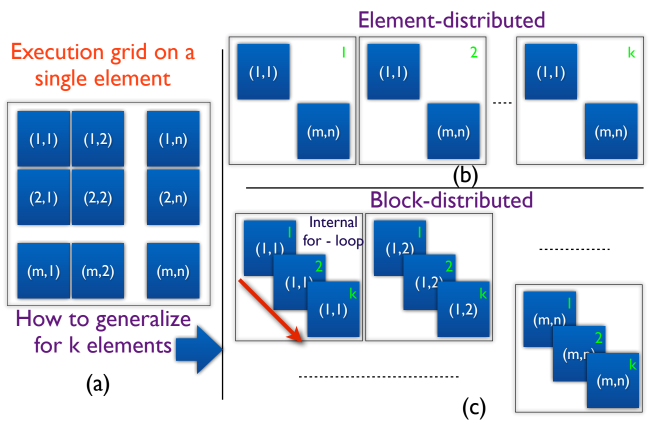


Fig. 5. Generalization of MPAs from parallel SPAs

Based upon the similarities between single element and multi-element processing, we propose two strategies to implement MPAs from existing SPAs: the per-block and the per-element generalization (Figure 5).

The per-block generalization (Figure 5(b) and Algorithm 5) is the simplest method to extend from single element processing which preserves the granularity per element. A typical parallel SPA requires the execution of multiple blocks, with each block processing one part of the input (Figure 5(a)). Our per-block generalization employs almost identical kernels except an additional internal for-loop over all elements. In

```
1:  Input : AoS input data, $n$ element size, $nE$ No. of elements
2:  Output: multi-element operation
3:  Call $MPA\_kernel <<< SPA\_cfg, null >>>(...)$
4:  function MPA_KERNEL(AoS data)
5:      $bid \leftarrow get\_blockId()$;
6:      __shared__ $s[n]$;
7:      for $i \leftarrow 1, nE$ do                ▷ Loop inside the kernel
8:          Upload $bid^{th}$block of $i^{th}$element to $s$;
9:          inline SPA algorithm;
10:     end for
11: end function
```

Algorithm 5: Per-block Parallelism MPA

other words, the method replaces the external CPU for-loop with the internal GPU counterpart. This method has several advantages: it is simple, able to exploit the existing optimization for SPAs, and capable of removing the call overheads. For its simplicity, per-block generalization is the quick solution for MPAs. However, in practice it is unlikely to be the most efficient method because the condition for its effective usage is similar to SPAs – namely large element size; however in that case the overhead is negligible and the overall speedup is small.

The per-element method generalizes the SPAs with a constraint that a single execution block perform the entire processing of a single element (Algorithm 6). In this case, the internal structure of a kernel resembles the decomposition of a SPA. Contrary to the per-block generalization, moving from single element processing to a per-element method is non-trivial. We may or may not reuse the existing GPU implementation since the resource constraint is the amount of memory per execution block rather than entire GPU system. Though it has limited resource support, the method turns out to be the most effective approach for small to medium element sizes. It is also scales well with the number of elements.

While a per-element method is generally better than a per-block generalization, there is a turning point where per-block becomes a better option. We determine this turning point based on the ability to utilize the parallel processing power of the system: the number of execution blocks. In the per-element strategy, it is the number of elements while in the per-block strategy it is equal to the number of data blocks per element. Our implementation then chooses the method that gives a higher number of execution blocks. This selection strategy is applied for small to medium problem sizes which is the main concern of our discussion. Again, our selection strategy depends both on the number of elements and the element size to achieve maximal processing throughput.

Next we discuss some of the essential techniques that we apply in our multi-element processing framework to achieve maximal performance.

```
1:  **Input** : input data in AoS format, $n$ element size, $nE$ No. of elements
2:  **Output**: multi-element operation
3:  $CallMPA\_kernel <<< nE, bl, NULL >>>$(...)
4:  **function** MPA_KERNEL(AoS data)
5:      $bid \leftarrow get\_blockId()$;
6:      __shared__ $s[n]$;
7:      **loop** $i \leftarrow first\_block$ to $last\_block$;
8:          Upload the $i^{th}$block of $bid^{th}$element to $s$;
9:          Process $s$ then output AoS data of the $i^{th}$block;
10:     **end loop**
11: **end function**
```

Algorithm 6: Per-element MPA kernel

E. Optimizations for multi-element processing

1) Execution block configuration: Kernel configuration (number of threads, number of blocks) determines the parallelism granularity and has strong influences on performance. While a tuning strategy is often required to maximize performance [49], it is important to start with a good estimation as the configuration space is large and displays non-linear behavior. A good estimator must satisfy two conditions: it scales well across platforms and it is adapted to the problem size. That means it has to take into account the hardware configuration of the running system (*i.e* number of registers, size of the shared memory) and the kernel information (*i.e.* number of threads, shared memory usage and problem size).

Our estimator employs a multi-level strategy. The first heuristic is based on occupancy calculations [16]. GPU memory offers very high bandwidth but also high latency. The occupancy number reflects the capability to hide this memory latency. We chose the occupancy calculation methodology as it takes into account both the information of the underlying system and the execution kernel, making it an estimator that scales well. This is also the choice employed by the Thrust library [27].

Even though high occupancy indicates a good parallelism efficiency, it does not necessarily directly correspond to a high performance. We observed significant variation in the performance of an algorithm even in the permissible configuration range of high occupancy. In order to narrow down the search space, we employ the second strategy based on the capability of the system to use high performance resources: the registers. The idea comes from an observation by Volkov *et al.* [49] that between all admissible setups, the configuration using a minimum number of threads enables a maximal number of registers per execution threads will run faster.

We propose a combined approach which starts with the minimal block size – the wavefront – and increase a minimal step (wavefront size) each time till the occupancy requirement is met. This is reversed to the maximal block size strategy employed by Thrust [27] which starts the search with the maximum block size. Our strategy prevents idle threads from being generated, and reserve computational resource for working threads. As shown in Table I, we achieve higher performance with our minimal block size strategy in comparison to the fixed size strategy [49] and maximal block size strategy [27].

Configuration	Occupancy	Runtime (μs)	Notes
768	100%	80.5	Thrust
512	100%	79.1	
256	100%	77.8	
192	100%	77.4	Ours
128	75%	83.9	
64	50%	119.7	Volkov

TABLE I
OUR INITIAL CONFIGURATION BASED ON MINIMAL ADMISSIBLE OCCUPANCY CALCULATION SHOWS BETTER PERFORMANCE THAN EXISTING APPROACHES.

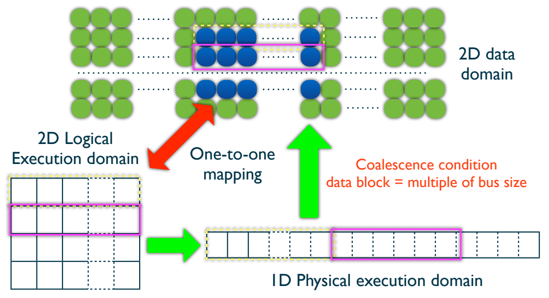

2D data domain
2D Logical Execution domain
One-to-one mapping
Coalescence condition data block = multiple of bus size
1D Physical execution domain

Fig. 6. The connection between data in 2D data domain to 2D execution block and the physical 1D execution thread on GPUs which determines the coalesced condition of 2D execution grid.

2) Mapping with a 2D grid: A 2D execution grid is often employed in image processing and matrix computation algorithms since it allows a one-to-one mapping from the execution grid to the processing space (Figure 6). One intriguing question is that how the 2D configuration might influence the performance of a kernel.

We evaluate the efficiency of a 2D configuration based on the coalesced condition. As the cache line has a 32-byte granularity, to achieve 100% bus utilization for a column-wise mapping we need at least 8 threads per row for single floating point (fp) computation and 4 for double (db). We validate and reconfirm this hypothesis using a 2D copy kernel with different configurations which yield the same number of threads. As shown in Table II, when the bus is under-utilized (one or two threads), accessing less data (with fp) is not faster than more data (with db); however when the bus is fully-utilized (eight threads and more) the transfer time with db is twice as large as with fp. The minimum number of threads per rows, eight for 4-byte data (fp) and four for 8-byte data (db) explains why in practice some configurations are generally preferred such as 8×8 or 16×16.

Based on the number of threads determined by the configuration estimator and the constraint of a 2D mapping, we propose an adaptive 2D configuration as follows:

- Compute the number of threads in 1D using the occupancy calculation.
- Choose the number of threads per row n as a multiplier of eight (floating) or four (double).
- Compute the number of rows m, giving preference to the configuration with $m = n$ for square domain computations.

Configuration	float	double	Note
128×1	0.7	0.69	float is not faster than double
64×2	0.37	0.36	
32×4	0.22	0.21	double-coalesced
16×8	0.13	0.20	float-coalesced
8×16	0.11	0.20	float is twice as fast as double
4×32	0.11	0.20	

TABLE II
PERFORMANCE OF DIFFERENT 2D BLOCK CONFIGURATION (DATA COPY FOR 256 MATRIX 128×128

3) Shared memory and double computation: Working with shared memory is an essential technique to extract maximum parallel processing power. Shared memory is an explicit programmable cache memory which has very low latency. Shared memory contents can be accessed in parallel through different memory banks. Bank conflicts happen when threads in the same wavefront accesses the same memory bank, in which case the action will be serialized. Regular one-to-one mappings often suffer from bank conflicts, for example, when threads access data row-wise. A typical technique to remove this bank conflict is to use padding.

Since each bank has a fixed width (typically 4-bytes) the amount of padding will differ based on the type of data. For example, the row-wise access with a padded array $s[32][33]$ suffers no bank conflicts with fp data but a 2-way bank conflict with db. One might try to remove this bank-conflict by storing the high word and low-word of a db separately on two different 32-bit arrays. However, this approach requires extra loading, storing and format conversion instructions. It turns out that these additions outweigh the benefits of 2-way bank conflict removal. Also note that db computations require higher arithmetic intensity than fp computations, so they will suffer less performance loss due to the shared memory bank conflict. We conclude that the same shared memory structures could be used for both fp and db computation. That facilitates the use of template programming to yield a unified code base for both fp and db computation.

4) Hybrid model based on SIMD-width: There are certain cases when the computation only requires a sub-block amount of memory and can not be handled by a single thread, for example the matrix vector multiplication with small ranks. While the block-based approach can handle the problem, we again face the resource inefficiency of a large element algorithm on small elements. This also leads to a large performance gap between thread-based execution and block-based execution results as shown on Figure 9.

We propose a hybrid approach which maps multiple threads— a group—to a single element, but the entire block is split amongst multiple elements. We allocate an equal number

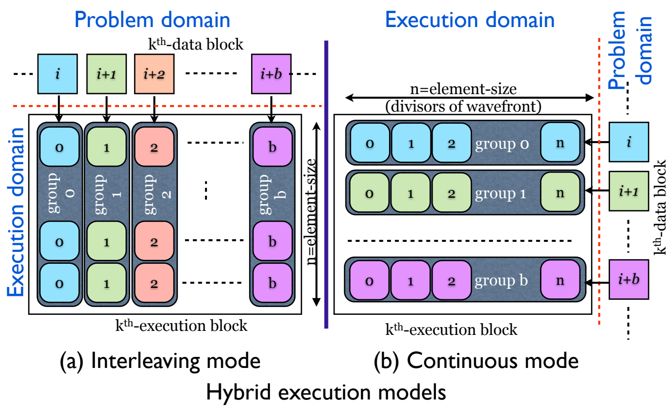


Fig. 7. Parallel execution mapping for MPAs based on a hybrid model, a transition between thread-based and block-based execution.

of threads—a group size—to an element and the shared memory is divided between groups—per element similar to the thread—based approach. The members of a group perform the computation cooperatively on shared memory. Here are two options to perform the computation: interleaving and contiguous groups.

In the interleaving option (Firgure 7(a)), two neighboring threads belong to two different element groups. This option is vital when the computation per thread is independent, as their is no need for synchronization between elements in the same group. The SoA data structure is used for this model as each thread per group works on contiguous elements in memory.

In practice, the computation is dependent and normally requires synchronization points similar to block-based methods. This might hurt the performance. Fortunately, the synchronizations can be effectively removed if all the group members belong to the same wavefront (SIMD width) - the contiguous groups (Figure 7(b)). This allow us flexible block sizes which are divisors of wavefront size, which is 32 on current GPU architecture. The AoS data structure is the preferred with this model as it allow cooperative works and data IOs between threads of the same group.

We observed a significant improvement on BLAS2 operations with small rank matrices when we apply the hybrid model. There exists data reuse between threads mapped to a given element; however, it is insufficient to allocate an entire block to that element. Therefore, it would be better to divide the resources of a single block to multiple elements just enough to amortize the per-element performance.

F. Optimal processing framework for large elements

While the focus of the paper is on "small" element processing, our framework also provides a solution for large elements. Based on the assumption of the uniformity of the data and operations, we exploit the optimal asynchronous framework for multi-image processing—the ISP—by Ha *et al.* [24] for MPAs with large elements. We further extend the idea for small elements when the memory requirements are larger than GPU main memory. In such a case, we divide our data into chunks, each including a number of small elements that can fit within GPU main memory. We then apply both the asynchronous streaming framework on our multiple chunks structure and our MPA for each chunk. Besides the capability to handle the problem with an arbitrarily large number of elements, the chunk structure also improves the GPU-CPU transfer as it reduces the data overhead and reaches the minimum size requirement for effective out-of-core bandwidth usage.

IV. Results

In this section, we report the results of implementing our multi-element framework in the context of linear algebra operations. The system we used in our experiments is a PC server, 16 Intel Xeon Core x5770 running at 2.93Ghz, 12-GB DDR3 1600, with four NVIDIA Tesla C2050 cards. The program is compiled with CUDA NVCC 4.2. Run-time of each function is measured in milliseconds.

We perform tests on synthetic data randomly generated to cover different ranges of element (matrix) size, number of elements and data type (float or double). While our framework is not limited to linear algebra functions, we found it more appropriate to compare our implementation to generally-used and highly-optimized packages such as NVIDIA's CuBlas and MAGMA. Note that currently CuBlas and MAGMA are starting to support batch mode processing; however their features do not cover all the functions such as BLAS Level 1 and BLAS Level 2. We compare our direct MPA implementation to the batch mode available with the CuBlas (MAGMA) and our batch processing model using multiple streams 2 .

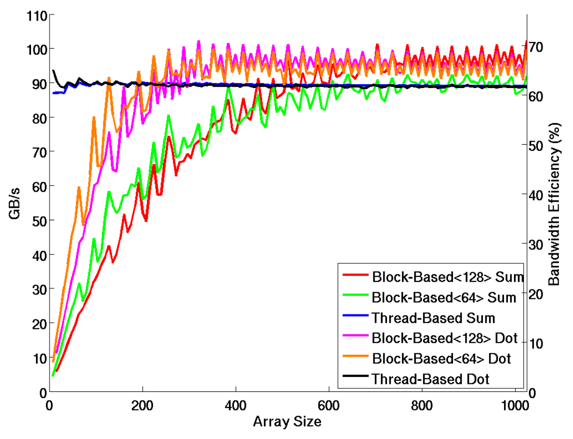
The tests were conducted by using a large array of structures (AoS) or structure of arrays (SoA) so that we nearly maximize the GPUs memory capacity. For our tests, we repeat the operations iteratively, effectively simulating many operations. The number of array operations was well over 10^7 in all cases. This ensures that the performance of the algorithm reaches its peak for a given input size.

Due to the nature of mapping computations to the GPU with thread blocks, there will be performance oscillations with respect to problem size for various BLAS operations. Peak performance is attained when the problem size is divisible by the block size. We demonstrate the performance of our framework for three canonical BLAS1, BLAS2, and BLAS3 operations: dot product of two vectors, matrix-vector multiplication, and matrix-matrix multiplication, respectively. The tests we show are only for double precision, although our framework supports single precision as well.

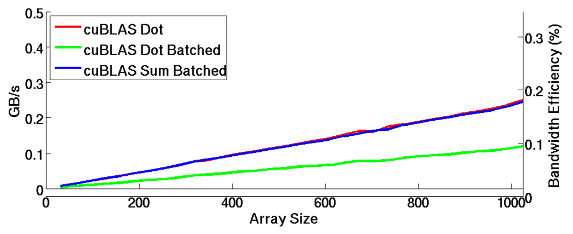
A. BLAS Level 1 functions

Figure 8 shows our result for BLAS Level 1 functions. As the computational complexity of these functions is low, their performance characteristics are constrained by memory bandwidth, so we use the bandwidth efficiency as its appropriate measurement. The results show the following:

- Our approaches were capable of reaching up to 65% of the theoretical bandwidth while other methods are hundreds of times slower.
- The thread-based model using the SoA data structure shows a steady performance which is higher than that of the block-based approach for tiny element sizes. However, when the element size increases (larger than 160), the latter, which allows cooperative computation, becomes faster.
- The performance variation with different block sizes (64 and 128) is significant even in the range of high occupancy configurations (up to 20 GB/s, 50% at a vector size of 160). The 64 threads configuration is faster. These observations confirm our discussions in the previous section about optimal block size strategy.
- CuBlas algorithms fail to perform well in batch streaming modes due to implicit synchronization as data was copied from GPU output to CPU memory.



(a)



(b)

Fig. 8. Performance results for BLAS1 functions with different configurations (64, 128 block size) and execution models (block-based, thread-based models as well as for-loop and batching model from cuBlas) demonstrating that our method can leverage performance and out-perform existing methods.

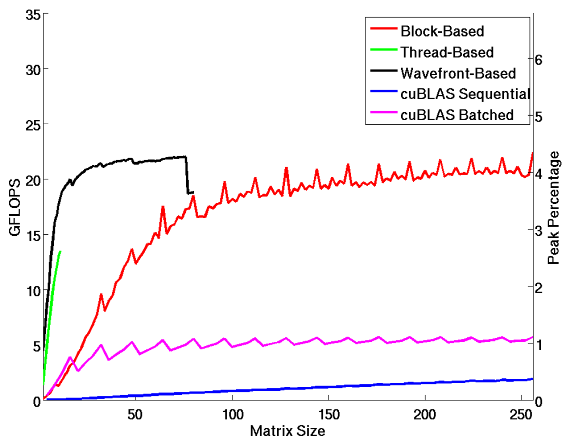


Fig. 9. Performance results for BLAS 2 functions. The hybrid approach is virtually capable of closing the performance gap between the thread-based and block-based models.

B. BLAS Level 2 functions

As illustrated on Figure 9, as the complexity increases, the performance gap between our methods and the batch mode is narrower: about four times higher over the range. The thread-based model performs well with tiny matrices; however, its range is limited to matrices of size less than 16×16. Our hybrid method helps increase the amount of shared memory per-element and exploits the parallelism of SIMD threads, hence it extends the range while providing solid performance gains. It closes the gap in performance between the thread-based and block-based approaches. Our simple batching method (Algorithm 2) shows significant improvement over the sequential-for-loop approach, and is comparable to an optimized CuBlas batch function.

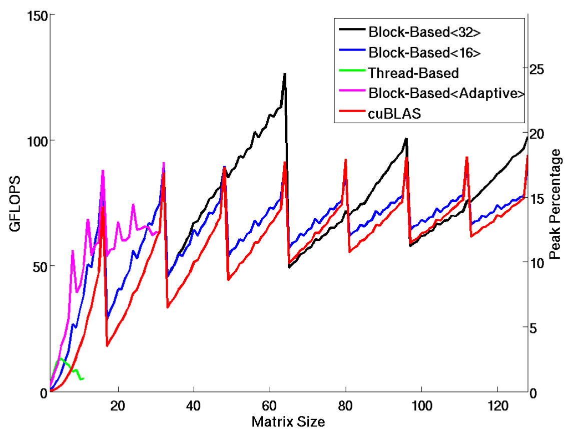


Fig. 10. Performance results for BLAS Level 3 functions, as the algorithms depend on block subdivision and have high computational intensity, our performance matches existing methods at the block size bounds while being significantly better with off-bound ranges.

C. BLAS 3 Level functions

BLAS Level 3 operations are compute-bounded and require a considerable amount of resources. The following observations can be made based upon the data presented in Figure 10:

- Our methods significantly improve over other methods for small sizes (from $2 \times 2 - 32 \times 32$).
- We provide competitive performance for large matrix sizes. Given that the memory and computational requirements increase algebraically, when the computational intensity is high enough to saturate the GPU computational power, we do not improve much over the CuBlas function running batch mode.
- Our adaptive method which chooses the model and block size based on number of elements and element sizes gives us competitive performance on the entire input range.

The performance of our execution models from BLAS Level 1, 2, and 3 show that we have addressed the problem of existing methods when dealing with small matrix sizes, and our models are capable of saturating the performance even with small element sizes.

D. LAPACK Functions

We compare our LAPACK functions with Cholesky and LU decomposition provided by MAGMA [43]. It is clear that our method outperforms the MAGMA solutions and is able to reach 50 GFLOPS at relatively small matrix sizes on the order of 120×120. Our performance curve is low at small sizes due to the fact that the our algorithm is generalized from the block subdivision strategy which performs poorly at sizes smaller than a single block. We envisage that improvements can be made in the decomposition to help eek out further performance gains at small matrix sizes.

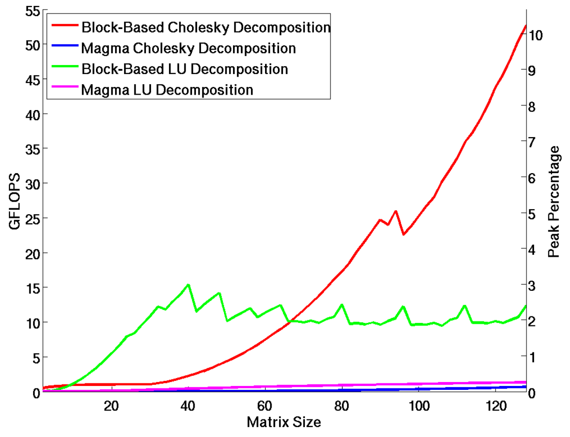


Fig. 11. Our experiments on batch LAPACK-type operations.

V. Summary and Conclusions

In this paper we present the algorithmic articulation of a framework designed to exhibit strong performance characteristics when applied to multi-element processing. The main contribution of our work is the demonstration of a framework that is specifically designed to overcome the two main deficiencies that exist within current multi-element processing frameworks. First, our framework has been designed to handle a range of element sizes from the very small to the moderately large. Second, our framework has been specifically constructed to address the aforementioned overhead issue. Instead of hiding the overhead, we remove it entirely. To demonstrate the efficacy of our approach, we have developed a GPU library for batch processing of linear algebra operations such as dot-product (BLAS1), matrix-vector multiplication (BLAS2), matrix-matrix multiplication (BLAS3) and linear system solving (*e.g.* Cholesky decomposition with backsolve).

We presented the results of using our system for a myriad of matrix ranks, from the very small to the moderately large – a range often neglected by many competing systems. We show that by exploiting prior information when we perform multi-element processing, we then can utilize the available massively SIMD streaming hardware more efficiently.

The framework we provide is not limited to only linear algebra operations. By its construction, it can be exploited for any multi-element processing operation. By adding the extra dimension provided by "batch" processing, we were able to more fully utilize the capabilities of available massively SIMD hardware. In future work, we envisage extending this framework to other important linear algebra operations (such as eigenvalue computation and singular value decomposition), as well as to other non linear algebra operations such as batch image processing.

Though our results have confirmed our hypotheses, we believe we have just open the door of multi-element processing problems. The optimal solutions for a particular problem is still a challenge, our framework only provide tools to set a quick, lower bound solution. A final solution might requires a combination of different strategies. And we see this as our future works.

References

[1] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov, *A hybridization methodology for high-performance linear algebra software for gpus*, in GPU Computing Gems, Jade Edition **2** (2011), 473–484.

[2] A.M. Alattar, *A probabilistic filter for eliminating temporal noise in time-varying image sequences*, ISCAS '92. Proceedings., vol. 3, May 1992, pp. 1491 –1494.

[3] ATI, *AMD Accelerated Parallel Processing OpenGL Programming Guide*, January 2011.

[4] Nathan Bell, Steven Dalton, and Luke Olson, *Exposing fine-grained parallelism in algebraic multigrid methods*, NVIDIA Technical Report NVR-2011-002, NVIDIA Corporation, jun 2011.

[5] Nathan Bell, Yizhou Yu, and Peter J. Mucha, *Particle-based simulation of granular materials*, Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation (New York, NY, USA), SCA '05, ACM, 2005, pp. 77–86.

[6] M. Bertero and P. Boccacci, *Super-resolution in computational imaging*, Micron **34** (2003), no. 67, 265 – 273.

[7] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley, *ScaLAPACK users' guide*, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[8] Guy E. Blelloch, *Nesl: A nested data-parallel language*, Tech. report, Pittsburgh, PA, USA, 1992.

[9] S. Borman and R.L Stevenson, *Super-resolution from image sequences – A review*, Proceedings of the 1998 Midwest Symposium on Circuits and Systems (Notre Dame, IN, USA), IEEE, August 1998, pp. 374–378.

[10] V.M. Bove, Jr., and J.A. Watlington, *Cheops: A reconfigurable data-flow system for video processing*, ITCS' 95, 1995, pp. 140–149.

[11] J.M. Boyce, *Noise reduction of image sequences using adaptive motion compensated frame averaging*, ICASSP' 92, Proceedings, vol. 3, Mar 1992, pp. 461–464.

[12] Ian Buck, *Gpu computing: Programming a massively parallel processor*, Proceedings of the International Symposium on Code Generation and Optimization (Washington, DC, USA), CGO '07, IEEE Computer Society, 2007, pp. 17–.

[13] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha, *Scan primitives for vector computers*, Proceedings of the 1990 ACM/IEEE Conference on Supercomputing (Los Alamitos, CA, USA), Supercomputing '90, IEEE Computer Society Press, 1990, pp. 666–675.

[14] AMD Corp, *AMD Accelerated Parallel Processing*, Dec 2011.

[15] ———, *AMD Accelerated Parallel Processing Math Libraries*, January 2011.

[16] NVIDIA Corp, *CUDA Occupancy Calculator*, Oct 2007.

[17] ———, *CUDA Programming Guide 4.2*, Apr 2012.

[18] B.C. Davis, P.T. Fletcher, E. Bullitt, and S. Joshi, *Population shape regression from random design data*, ICCV'07 (2007), 1–7.

[19] M.O. Deville, E.H. Mund, and P.F. Fischer, *High order methods for incompressible fluid flow*, Cambridge University Press, 2002.

[20] Jack Dongarra and Shirley Moore, *Empirical performance tuning of dense linear algebra software*, Chapman & Hall, 2010.

[21] F. Dufaux and F. Moscheni, *Motion estimation techniques for digital tv: a review and a new contribution*, IEEE, Proceedings **83** (1995), 858 –876.

[22] M. Goesele, N. Snavely, B. Curless, H. Hoppe, and S.M. Seitz, *Multi-view stereo for community photo collections*, ICCV' 07, Oct 2007, pp. 1 –8.

[23] William Gropp, Rajeev Thakur, and Ewing Lusk, *Using mpi-2: Advanced features of the message passing interface*, 2nd ed., MIT Press, Cambridge, MA, USA, 1999.

[24] Linh Khanh Ha, Jens Kruger, Joao Luiz Dihl Comba, Claudio T. Silva, and Sarang Joshi, *Isp: An optimal out-of-core image-set processing streaming architecture for parallel heterogeneous systems*, IEEE Transactions on Visualization and Computer Graphics **18** (2012), 838–851.

[25] Mark Harris, John Owens, Shubho Sengupta, Yao Zhang, and Andrew Davidson, *CUDPP: CUDA data parallel primitives library*, http://www.gpgpu.org/developer/cudpp/, 2007.

[26] J. Hays and A.A. Efros, *Scene completion using millions of photographs*, TOG **26** (2007).

[27] Jared Hoberock and Nathan Bell, *Thrust: A parallel template library*, 2010, Version 1.3.0.

[28] T. J. R. Hughes, *The finite element method: linear static and dynamic finite element analysis*, Prentice-Hall, 1987.

[29] Intel, *AMD Core Math Library. Version 5.1.0*, Oct 2011.

[30] ______, *Intel Math Kernel Library Reference Manual* , Oct 2011.

[31] G.E. Karniadakis and S.J. Sherwin, *Spectral/hp element methods for cfd - 2^{nd} edition*, Oxford University Press, UK, 2005.

[32] Robert M. Kirby, Bernardo Cockburn, and Spencer J. Sherwin, *To cg or to hdg: A comparative study*, Journal of Scientific Computing **51** (2011), 183–212.

[33] Robert M. Kirby and George Em Karniadakis, *Spectral element and hp methods*, John Wiley & Sons, Ltd, 2004.

[34] Rajib Nath, Stanimire Tomov, Tingxing "Tim" Dong, and Jack Dongarra, *Optimizing symmetric dense matrix-vector multiplication on gpus*, Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (New York, NY, USA), SC '11, ACM, 2011, pp. 6:1–6:10.

[35] Rajib Nath, Stanimire Tomov, and Jack Dongarra, *Accelerating gpu kernels for dense linear algebra*, Proceedings of the 9th international conference on High performance computing for computational science (Berlin, Heidelberg), VECPAR'10, Springer-Verlag, 2011, pp. 83–92.

[36] NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara 95050, USA, *CUDA C Best Practices Guide*, 4.1 ed., May 2012.

[37] OpenMP Architecture Review Board, *Openmp application program interface*, Specification, 7 2011.

[38] J.D. Owens, M. Houston, D. Luebke, S. Green, J.E. Stone, and J.C. Phillips, *Gpu computing*, Proceedings of the IEEE **96** (2008), no. 5, 879–899.

[39] S Preston, L.K. Ha, and S. Joshi, http://www.sci.utah.edu/software.html, AtlasWerks: High-performance tools for diffeomorphic 3D image registration and atlas building.

[40] S. Rixner, W.J. Dally, U.J. Kapasi, B. Khailany, A. López-Lagunas, P.R. Mattson, and J.D. Owens, *A bandwidth-efficient architecture for media processing*, MICRO 31th. Proceedings., 1998, pp. 3–13.

[41] G. Ruetsch and P. Micikevicius, *Optimizing Parallel Reduction in CUDA*, 2009.

[42] Ch. Schwab, *p- and hp- finite element methods: Theory and applications to solid and fluid mechanics*, Oxford University Press (USA), 1999.

[43] Brian J. Smith, *R package magma: Matrix algebra on gpu and multicore architectures, version 0.2.2*, August 27, 2010, [On-line] http://cran.r-project.org/package=magma.

[44] N. Snavely, S.M. Seitz, and R. Szeliski, *Photo tourism: Exploring photo collections in 3D*, SIGGRAPH'06, Proceedings, 2006, pp. 835–846.

[45] F. Song, S. Tomov, and J. Dongarra, *Efficient support for matrix computations on heterogeneous multi-core and multi-gpu architectures*, (2011).

[46] R.L Stevenson and S. Borman, *Image sequence processing*, Encyclopedia of Optical Engineering, Marcel Dekker, New York, September 2003, pp. 840 – 879.

[47] B.A. Szabó and I. Babuška, *Finite element analysis*, John Wiley & Sons, New York, 1991.

[48] Robert A. van de Geijn, *Using PLAPACK: parallel linear algebra package*, MIT Press, Cambridge, MA, USA, 1997.

[49] Vasily Volkov and James W. Demmel, *Benchmarking gpus to tune dense linear algebra*, Proceedings of the 2008 ACM/IEEE conference on Supercomputing (Piscataway, NJ, USA), SC '08, IEEE Press, 2008, pp. 31:1–31:11.

[50] H. Wang, Y. Wexler, E. Ofek, and H. Hoppe, *Factoring repeated content within and among images*, TOG **27** (2008), 14:1–14:10.

[51] C. Whaley, A. Petitet, and J.J. Dongarra, *Automated empirical optimization of software and the atlas project*, PARALLEL COMPUTING **27** (2000), 2001.