ISP: An Optimal Out-Of-Core Image-Set Processing Streaming Architecture for Parallel Heterogeneous Systems

Linh K. Ha, Jens Krüger, João L. D. Comba, Cláudio T. Silva, Sarang Joshi.

Abstract—Image population analysis is the class of statistical methods that plays a central role in understanding the development, evolution and disease of a population. However, these techniques often require excessive computational power and memory that are compounded with a large number of volumetric inputs. Restricted access to supercomputing power limits its influence in general research and practical applications. In this paper we introduce ISP, an Image-Set Processing streaming framework that harnesses the processing power of commodity heterogeneous CPU/GPU systems and attempts to solve this computational problem. In ISP we introduce specially-designed streaming algorithms and data structures that provide an optimal solution for out-of-core multi-image processing problems both in terms of memory usage and computational efficiency. ISP makes use of the asynchronous execution mechanism supported by parallel heterogeneous systems to efficiently hide the inherent latency of the processing pipeline of out-of-core approaches. Consequently, with computationally intensive problems, the ISP out-of-core solution can achieve the same performance as the in-core solution. We demonstrate the efficiency of the ISP framework on synthetic and real datasets.

Index Terms—GPUs, out-of-core processing, atlas construction, diffeomorphism, multi-image processing framework

1 Introduction

Image-set processing is an advanced image processing technique, widely used in medical imaging [15], video processing [5], [41], astronomy [1], [6], visual robot control [18], virtual reality [44], [45], modeling and reconstruction [24], [28], among others. This technique provides an extended processing power unattainable with single-image processing techniques. For example, for images captured by low-end devices, an image-set average operation is capable of reducing noise without compromising details, as well as increasing bit depth and quality [1], [6]. Another example is the reconstruction of the 3D structure of large buildings or monuments. Using multiple images captured from different points of view and under different lighting conditions, or from public databases such as Flicker or Google Images, it is possible to reconstruct the 3D layout of a large scene or the interior of a building [24], [28], [44], [45]. Image-set processing is also essential for real-time video processing, with applications in video compression, computer vision, automated obstacle avoidance vehicles, video surveillance and security control.

Image-set processing provides the backbone for Computational Anatomy, an important tool in analyzing populations composed of hundreds to thousands of subjects. In this paper, we use the atlas construction technique as a driving challenge and also an illustration for the effectiveness of our approach. The method plays a central role in computational anatomy, particularly in understanding the variability of brain anatomy [12], [15], [22], [34]. As illustrated in Figure 1, the atlas construction allows the analysis of a population using age regression over average brain atlases computed at different ages.

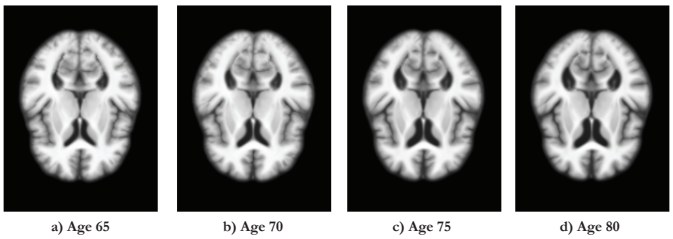


Fig. 1. Age regression analysis on the ADNI dataset by computing the average brain atlases at different ages (65, 70, 75, and 80) confirms the proposition that fluid space is larger because the brain atrophies over time.

However, the benefits of using image-set processing techniques over single-image processing brings major computational challenges. First, they involve a huge amount of data that easily exceeds the direct processing capability of the system. Second, they demand massive amounts of computation, which often require days or even months to complete. As a result, image-set techniques often call for super-computing systems [12] or large-scale clusters [25], [44] that limit their applications to large laboratories. Based on the commodity hardware, a solution will make this technique available to a widespread audience with limited computational

- *Linh K. Ha and Sarang Joshi are with the Scientific Imaging and Computing Institute, University of Utah.*
- *Jens Krüger is with the University of Saarland.*
- *João L. D. Comba is with the Federal University of Rio Grande do Sul (UFRGS-Brazil).*
- *Cláudio T. Silva is with Polytechnic Institute of NYU (NYU-Poly).*

resources, thus increasing its use in many existing problems.

In this paper, we discuss a solution for image-set processing problems on commodity hardware using graphic processing units (GPUs) combined with an out-of-core streaming model. The contributions of our paper are:

- A high-performance, image-set processing framework with a proof-of-concept optimal streaming model;
- The definition of the basic building blocks of a general framework, which allows an efficient-implementation of image-set algorithms;
- The introduction of concepts for implicit and explicit pipelining, with considerations regarding their efficiency;
- A pseudo-loss-less compression scheme for floating point inputs, which allows simple, effective, and high performance encoding and decoding schemes;
- An analysis of the reasons for streaming degeneracy, and a solution based on an order-independent model;
- Solutions for cross-stream synchronisation to handle the profiling issue of asynchronous programming;
- A mechanism to extend the streaming model on heterogeneous systems;
- Guidelines based on the performance analysis that help developers profile their code performance and make quantitative decisions.

2 Related work

In the last decade, there is an emerging trend in the High Performance Computing (HPC) community to use heterogeneous processing systems (e.g cell processors, FPGAs, multi-core GPUs, etc) to replace the conventional supercomputing model. Super-computing systems based on the heterogeneous model have been successfully exploited in some of the fastest computing systems, such as the current number one super-computer Tianhe-1A, the first computing system to achieve 2.5 petaflop/s [38].

The GPU processing model, with hundreds of simple and computation-centric processing cores, has been proved to be highly scalable to many problems. In particular, they are suitable to problems that can be easily computed in parallel, such as the ones that arise in image-processing applications.

Modern GPUs can offer a few Tera-flops of peak performance per unit, providing processing power equivalent to a super-computer in the mid-90s, while being considerably more cost-and energy-efficient. The huge in-core memory bandwidth, which has historically doubled every two years, is another advantage of GPU systems over the conventional processing model, adding substantial speed increases to the GPU-centric processing model. There are a number of image-processing applications implemented on GPUs [19], [42], [46] , most of which achieve from 20x to several magnitudes of speedup over CPU counterparts. Conceptually, our streaming framework is an extension to the idea of the fast GPU image-processing framework by Ha *et al.* [25], [26]. Their method achieved 60x speed up in comparison to an optimized, fully-parallel version running on an eight-core Xeon server for Greedy Iterative Diffeomorphic Atlas construction problem.

While the use of GPUs appears to be a good solution to the computational requirements of image-set processing techniques, the large memory footprint remains an open problem. Even though the memory bandwidth is fast enough, the size of the on-board GPU memory is very limited. Since GPU programs can only access on-board memory, data must be stored on the GPU when they are required. This incurs in some memory management scheme or out-of-core solution.

There are three primary approaches to out-of-core programming. The first is to use virtual memory based on operating system support. It is simple and unified for both in-core and out-of-core processing. However, due to a lack of application-specific knowledge about the data dependence and parallelism, this method often leads to a poor performance [47]. The second approach is to use compiler directed I/O to convert a program from in-core to out-of-core [4], [7], [39]. For programs with complicated data dependencies this approach is not as effective as the third approach that we use here: explicit I/O controls by developers. These methods concentrate on techniques to improve the cache coherency such as *caching* and *prefetching* [2], [9], [11], [31], [37] to reduce the I/O necessary for blocks already in main memory and/or by overlapping I/O operations with main-memory computations. Such methods exploit particular computational properties of each individual problem as part of the algorithm design. While explicit I/O controls are mostly application-specific, our method applies to a wide class of applications such as out-of-core image-set processing.

Out-of-core processing on GPUs is not a new technique, especially for scientific visualization [16], [21], [29], [48]. The amount of data generated in scientific simulations or modern data acquisition systems (e.g. laser scans) are reaching petascale sizes, which can easily overwhelm the direct visualization capability of any visualization system [49]. To allow real-time rendering, most out-of-core visualization techniques use some caching scheme to minimize the amount of data pushed through the system [2], [20], [43]. The advance of GPU hardware allowed the extension of such techniques to use acceleration structures, such as KD-trees or BVH trees [36], [50]. Hou *et al.* [30] addressed the problem of building the out-of-core BVH tree using a CUDA solution that is able to handle the number of input triangles several times larger than previous GPU algorithms. Nevertheless, none of the above techniques handle the problem of processing the entire data in real-time, but instead explore the spatial coherence to extract subsets of the data. In this paper, we handle out-of-core processing problems for image-set processing which requires the entire data to be processed. Our technique is orthogonal to existing out-of-core visualization techniques, and can also be used to speed up these algorithms.

Our out-of-core strategy exploits two key performance concepts: *prefetching and data-transfer-hiding* based on an asynchronous streaming execution model. Asynchronous processing is a pipeline-concurrent execution model that exploits the availability of multiple execution units in the system to run independent tasks concurrently. This strategy reduces idle stages and increases resource utilization. It can also hide data transfer by prefetching data. When a processing unit finishes the current task, it can start subsequent tasks without delay. In many circumstances, using this model significantly increases the overall system throughput.

Asynchronous processing is accomplished using streaming models for both tasks and data. Streaming is an efficient model for parallel processing, in which a task is divided into smaller entities to allow parallel executions. A stream is an abstraction of an execution unit; in particular, it represents a sequence of commands that are executed or accessed in a particular order. Pure data streams determine data parallelism processing model, while pure task streams determine the task parallelism model. In practice, a stream may be data-based, task-based, or a mixture. The only restriction in a stream is the execution order that is satisfied by a sequential consistency model [35], which makes a stream equivalent to a synchronous process. Different streams, on the other hand, may execute their commands out-of-order with respect to each other.

3 Image-Set Processing Operators

An image-set algorithm involves several image-set operations, most of which are extensions of single-image processing operations iterated over all input images. We use the atlas construction algorithm (Algorithm 1) [26] to illustrate this point. We build the image-set processing framework upon the single-image high-performance multi-scale processing framework proposed by Ha *et al.* [25] to exploit the optimized performance of the existing framework.

We define the image-set processing framework using a construction method that builds regular image-set operators from basic building blocks. This strategy allows

1: **Input** : N volume inputs
2: **Output**: Template atlas volume
3: **for** $k = 1$ to max_iters **do**
4: Fix images I_i^k, compute the template $\hat{I^k} = \frac{1}{N}\frac{\sum_{i=1}^{N} I_i^k w_i}{\sum_{i=1}^{N} w_i}$
5: **for** $i = 1$ to N **do** {loop over the images}
6: Fix the template $\hat{I^k}$, solve pairwise-matching problem between I_i^k and $\hat{I^k}$
7: Update deformed image I_i^k with current velocity
8: **end for**
9: **end for**

Algorithm 1: Atlas construction framework

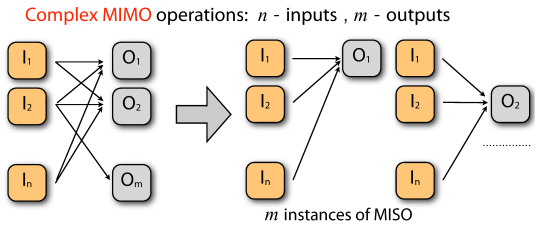


Fig. 2. General Multi-Input Multi-Output operators

a fine-grained and multi-level control of parallelism. It provides different execution strategies to be applied in each implementation, and therefore, to make a better use of the available resources. Here, we classify basic image-set operators into two main groups based on Flynn's taxonomy [23]: the Multiple-Input-Multiple-Output operators (MIMO) and the Multiple-Input-Single-Output operators (MISO).

The basic MIMO operators (Algorithm 2) are defined as functions where the n-th output image depends only on the n-th input image(s). These functions are the most frequently used in image-set processing, as they are direct extensions of single-image operations. Examples of these operations include adding, shifting, scaling, smoothing, filtering, de-noising images, and normalizing the intensity range.

The MISO operators (Algorithm 3) produce either one or just a small number of outputs. Examples include the computation of an average image, image energy, cross-correlation, cross-product of images, and finding the maximal and minimal values.

The implementation of general image-set operators is based on a decomposition strategy that breaks a complex function into multiple, basic operations. For example, consider a general MIMO function with the number of outputs M different from the number of inputs N and the k-th image to be output depends on multiple inputs. This operation can be implemented as M instances of a MISO operator (Figure 2).

Another group of frequently used image-set operators is the sliding-window operator (Figure 3.a). This operator computes an output image based on all values in a fixed-size sliding window of the input. This window moves as we compute the next output image. As shown on Figure 3.b, if we keep an input buffer with the size of the sliding window, as the window moves, we need to replace an entry of the window with the new

1: **Input** : N input images
2: **Output**: N processed output images
3: **for** $k = 1$ to N **do**
4: Upload the k-th image from the storage device to the processing device
5: Process the input in-core on the processing device
6: Download the output image back to the storage device
7: **end for**

Algorithm 2: Synchronous out-of-core MIMO operators

1: **Input** : N input volumes
2: **Output**: few numbers(sum, max/min, etc) or few images
3: **for** $k = 1$ to N **do**
4: Upload the k-th image from the storage device to the processing device
5: Process the input in-core on the precessing device
6: Update the accumulated output buffer on the processing device
7: **end for**
8: Write the final output to the storage device

Algorithm 3: Synchronous out-of-core MISO operators

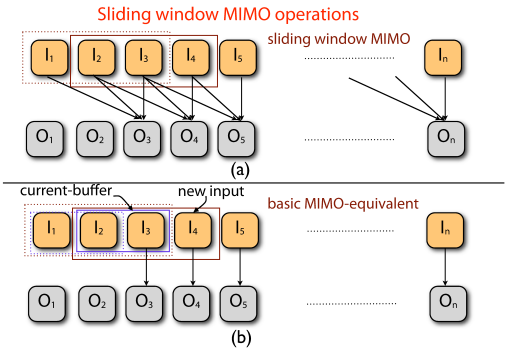


Fig. 3. Sliding window MIMO operators. (a) sliding window MIMO, (b) basic MIMO equivalent.

input data. In other words, the computation of a current output requires only a single input. Algorithmically, it is equivalent to the basic MIMO model. Overall, we can implement arbitrarily complex image-set functions based on the basic MIMO and MISO functions. We focus our discussion on how to efficiently implement these operators out-of-core.

The framework of Ha *et al.* [26] already has support for image-set and large data processing through a GPU-cluster implementation using MPI. It also offers a multi-GPU implementation to exploit available computing resources and to increase the amount of in-core GPU memory on a single processing node. However, both approaches are limited by the total amount of system memory. The out-of-core approach we introduce here has no restrictions on data input and can process the entire 3D-image brain dataset in a PC desktop equipped with commodity GPUs.

We also offer a more flexible solution to existing methods with two levels of streaming operations: out-of-core GPU combined with in-core-CPU, and fully out-of-core. The former utilizes the availability of the larger CPU memory system; in some cases the CPU (but not the GPU) memory may be sufficient for the entire computation. In the latter case, data that does not fit into CPU memory is transferred across two memory levels: from disk to CPU memory, and from CPU memory to GPU memory. We show that our streaming strategies could be generalized through multiple memory hierarchy levels. In the following discussion, GPUs are processing devices in the first out-of-core level; consequently, in-core memory refers to the GPU global memory, while the CPU memory plays the role of storage devices.

4 ISP Out-of-core Framework

We use synchronous implementations of the MIMO and MISO operators (Algorithms 2 and 3) as references for the correctness and performance improvement of our asynchronous implementations. We compare three different models to implement out-of-core image-set operations: implicit, hardware-aware, and hardware-independent. We show that the proposed strategies are optimal. Before that, we discuss the best achievable performance of an asynchronous algorithm.

4.1 Asynchronous optimal performance analysis

We base the analysis on two main assumptions. The first one is that image-set operators are order-independent, and therefore return the same results regardless of the order of the input data. The order-independence assumption allows us to define a processing order which can be different from the order given as input. The second assumption is that all images have similar sizes and therefore require the same amount of running time. While the former is satisfied with regular image-set functions, the latter is normally fulfilled with pre-processing the image-set data. Though our analysis is directly applied for the first level of out-of-core models, for the sake of the simplicity and clarity of the illustrations, the same analogy is applied for higher levels of out-of-core processing.

To evaluate the performance, we use a typical hardware configuration with three components: one computational unit (GPU) and two data transfer units (one for uploading, another for downloading data). In the performance analysis we use the following notation:

- n number of input images and n_s number of execution units;
- $\tau_{i,j}$: run-time of the i-th execution unit on the j-th input image;
- T_s, T_a : total synchronous/asynchronous processing time;
- T_u, T_e, T_d: uploading, executing, and downloading run-time per image;
- $\mathcal{T}_i$ total time spent by the execution unit i;
- $\mathcal{T}_u = n \times T_u$,$\mathcal{T}_e = n \times T_e$,$\mathcal{T}_d = n \times T_d$: total time spent in the upload, execution and download processes;
- $\mathcal{T}_{max} = max(\mathcal{T}_1, \mathcal{T}_2, \cdots \mathcal{T}_{n_s})$ maximum time spent by a single execution unit.

We determine the optimal asynchronous run-time, which we use as a reference to evaluate the efficiency of the proposed implementation method. In the ideal case, all execution units run independently in parallel. However, since each corresponds to a single execution entity, they perform tasks in sequential order. The total time an execution unit spends is $\mathcal{T}_i = \sum_{j=1}^{n} \tau_{i,j}$, which equals $n \times \tau_i$ (where τ_i is the run-time of the i-th stream

on a single-image). Since a image-set operation is only completed when all the execution units have completed their tasks, the run-time of the entire operation is at least $\mathcal{T}_{max} = max(\mathcal{T}_1, \mathcal{T}_2, \cdots \mathcal{T}_{n_s})$ or $T_a \geq \mathcal{T}_{max} = n \times \tau_{max}$. This is the optimum run-time that a system can achieve. Note that with this hardware configuration, the maximum time spent by an execution unit is given by $\tau_{max} = T_{max} = max(T_u, T_e, T_d)$.

4.2 Asynchronous processing benefit

We estimate the performance benefit of an asynchronous model, called by r_a, as the ratio of the synchronous processing run-time over the asynchronous run-time as:

$$r_a = \frac{T_s}{T_a} = \frac{\sum_{i=1}^{n_s} \mathcal{T}_i}{\mathcal{T}_{max}}. \quad (1)$$

As defined before, $\mathcal{T}_{max} = max(\mathcal{T}_1, \mathcal{T}_2, \cdots \mathcal{T}_{n_s})$, which allows us to conclude that: $r_a \leq n_s$. We call $r_p = n_s$ the potential performance benefit.

There are three interpretations we can draw from this conclusion. First, for any processing model, the practical speed up can not be higher than the number of streams or concurrent tasks which can be physically performed by the system. For example, on the system with only one data transfer unit and one computational unit, we expect a speed up ratio lower than 2. Second, the equality happens when $\mathcal{T}_{max} = \mathcal{T}_1 = \mathcal{T}_2 = \ldots = \mathcal{T}_{n_s}$, or the system is balanced. In other words, we require load balancing to achieve the maximum performance benefit out of the system. Third, as we increase the number of concurrent streams, we push the potential performance benefit r_p of a model even higher. That is, as we further subdivide the workload, we provides more opportunities for the system to optimize the program and reduce the maximum run-time $\mathcal{T}_{max}$. The performance favors models with a higher number of streams.

We analyze next three different streaming models: one implicit streaming model and two explicit approaches (Algorithms 5 and 6). We demonstrate how these performance strategies influence our streaming model design.

4.3 Implicit Streaming Model

The *implicit streaming model* (Algorithm 4) is solely based on data parallelism. It assigns each image to a stream, which works as a logical execution unit that performs the entire processing pipeline (Figure 4). As streams operate on different memory spaces, the data transfer on a stream can overlap with processing tasks on other streams.

streaming models (*hardware-aware* and *hardware-independent*) which depend on task parallelism. The former maps each hardware execution unit to a single stream, while the latter delineates a stream to a fixed function.

Figure 4 illustrates the execution of an implicit streaming model for a MIMO problem (Algorithm 4). It can be seen that when the number of images is significantly larger than the number of execution units, the overall

1: **Input** : N input volumes
2: **Output**: N processed output volumes
3: **for** $k = 1$ to N **do**
4: Load the data $iImg[k]$ from storage device to processing device, d_k on the stream k-th
5: **end for**
6: **for** $k = 1$ to N **do**
7: Apply the operator on data $d_o = oper(d_k)$ on the stream k-th
8: **end for**
9: **for** $k = 1$ to N **do**
10: Write output d_o to the storage device $oImg[k]$ on the stream k-th
11: **end for**

Algorithm 4: Implicit pipelining MIMO operator

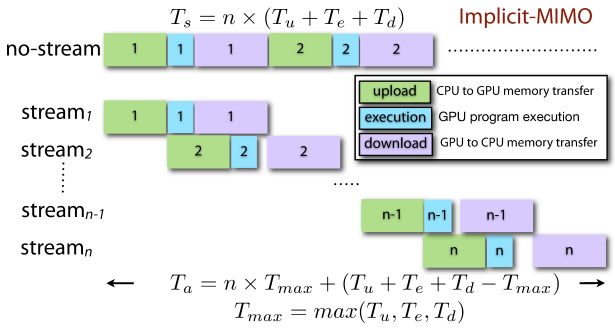


Fig. 4. Implicit processing model for MIMOs

processing time is approximately $n \times T_{max}$, which is the optimal run-time of asynchronous processing.

Although in the implicit model a higher benefit can be achieved by having more streams, our analysis demonstrated that this benefit has an upper bound on the number of hardware execution units in the system. Moreover, the mapping from implicit stream to the real hardware execution depends entirely on the system scheduler. The optimal mapping is non-trivial; in fact, it is an NP-hard problem. This helps to explain the difficulty of achieving high performance with the implicit model.

4.4 Hardware-Aware Streaming Model

The hardware-aware streaming model, as the name reflects, is based on the underlying system hardware, in which exists a one-to-one mapping between streams defined by the program and the real execution hardware in the system. For example, assuming that the system allows parallel data uploads and downloads using separated DMAs, there are three streams mapping to three

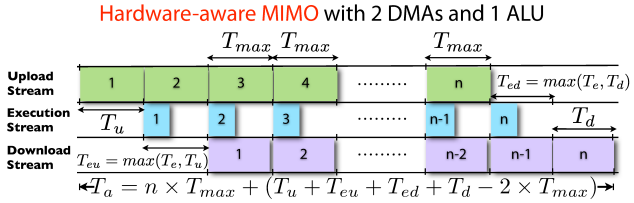


Fig. 5. Pipeline processing model for Multiple Images Multiple Output

Input : N input volumes, device input buffers $d_i[3]$ and device input buffers $d_o[3]$
Output: N processed output volumes
for $k = 1$ to $N + 2$ **do**
  if $k <= N$ **then**
    Load the data $iImg[k]$ from storage device to device buffer $d_i[k\%3]$ on the **upload** stream
  end if
  if $k > 1$ and $k - 1 <= N$ **then**
    Apply the operator on device buffer $d_o[(k-1)\%3] = oper(d_i[(k-1)\%3])$ on **execution** stream
  end if
  if $k > 2$ and $k - 2 <= N$ **then**
    Write output $d_o[(k-2)\%3]$ to the storage device $oImg[(k-2)]$ on the **download** stream
  end if
  Synchronize streams
end for

Algorithm 5: Explicit pipelining MIMO operator

Input : N input volumes, device input buffers $d_i[2]$ and device input buffers $d_o[2]$
Output: few numbers(sum, max/min, etc) or few images
for $k = 1$ to $N + 1$ **do**
  if $k <= N$ **then**
    Load the data $iImg[k]$ from storage device to device buffer $d_i[k\%2]$ on the **upload** stream
  end if
  if $k > 1$ and $k - 1 <= N$ **then**
    Apply the operator on device buffer $d_o[(k-1)\%2] = oper(d_i[(k-1)\%2])$ on **execution** stream
  end if
  Store/Accumulate result on processing device
  Synchronize streams
end for

Algorithm 6: Explicit pipelining MISO operator

execution devices. The execution of this model for MIMO problems is illustrated in Figure 5. A timing analysis of the method shows that the processing time in this case is also optimal. Because the hardware-aware model reflects the actual execution of asynchronous processes in the system, it requires developers to provide prior information about the architecture of the underlying system. In other words, it requires a different implementation on different hardware. The performance benefit of this model is limited by the number of streams or hardware devices defined by the program.

4.5 Hardware-Independent Streaming Model

The last processing strategy, the hardware-independent model, is a generalization of the hardware-aware model. Instead of decomposing tasks based on the actual hardware configuration, we assume that there exists one special execution unit for every task, and that we can

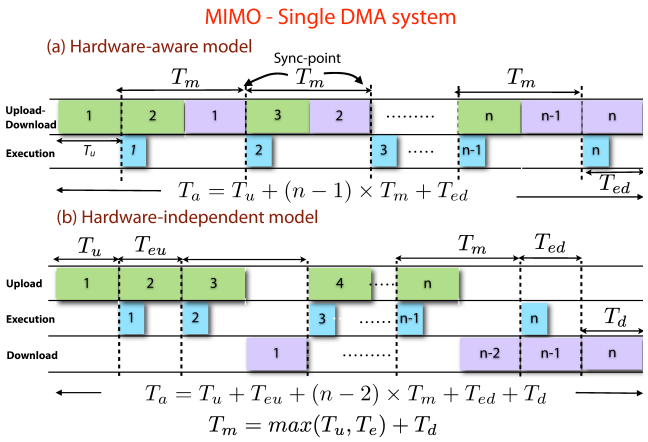


Fig. 6. Although the hardware-independent model incorrectly predicts the system configuration, the performance is still optimal.

assign each task a single stream. In the case of MIMO operations, there are three primary tasks to apply for each image: data upload, processing, and data download. A system with two data transfer units and one processing unit results in a streaming scheme similar to the hardware-aware model; consequently, this model also achieves the optimal run-time.

Usually there are more tasks than the actual number of execution units. For example, in a commodity system there is only one DMA to perform data uploading and downloading. In this case, it is possible that several tasks are mapped to the same execution unit (e.g. data upload and download can map to the same unit). The question is how efficient it can be when it incorrectly predicts the underlying system–in particular, when there are multiple streams sharing the same execution unit.

Data independence results in no performance loss, as the system can instantly switch between tasks. This function is performed automatically, since the shared information is available only at the system level. Figure 6 illustrates the run-time analysis of an optimal solution for a MIMO operation in a system with one DMA and one ALU using the hardware-aware and hardware-independent implementations. It shows that the hardware-independent model incorrectly predicts the underlying execution system, but still performs optimally.

4.6 Discussion on streaming modes

The primary advantage of the implicit approach is that developers are relieved from the burden of asynchronous scheduling. Furthermore, the stream has the same execution flow as when processing a single image. No further changes or synchronization is needed since each stream works on different data. However, it has several disadvantages:

- The implicit model does not reduce the memory usage and all the data must be loaded in-core. Hence, it can't be used for out-of-core processing;
- It requires the decomposition of input data and combination of output results, which is not always possible;

- Although automatic scheduling hides executions from developers, understanding the physical execution is essential to profile the performance and to estimate the benefit of the method (an estimation that is an important factor for making optimization decisions);
- The performance efficiency of the implicit streaming model is largely dependent on the scheduling algorithm used by the operating system or the concurrent controller. In fact, the optimal scheduling problem is NP-hard. This explains why, in practice, this approach does not always provide the predicted optimal performance;
- The implicit model has an order-dependency that limits the execution of streams. In particular, streams execute in the same order of the logical flow: uploading-processing-downloading. However, reordering is an effective strategy to handle degenerate cases, including synchronous functions calls.

Most of the weaknesses of the implicit model can be handled by explicit approaches:

- Explicit methods require a lower memory footprint $O(C)$, compared to $O(N)$ of the implicit model. The number of memory blocks is either equal to the number of hardware devices in the hardware-aware model, or to the number of decomposed tasks in the hardware-independent model. Therefore, they are suitable for out-of-core processing;
- As it is always possible to divide an out-of-core algorithm into three primary tasks, it is easier to decompose tasks than partition data;
- The explicit method uses an explicit scheduler to control execution. This allows developers to profile the performance of their code before execution. It also reduces the complexity of the scheduling problem to a trivial mapping, so it is optimal even without any automatic scheduler support. And finally, it helps to understand why degeneracy happens, how it affects performance, and how to handle it.

Between the two explicit approaches, the independent model is preferred in practice over the hardware-aware for a number of reasons. The hardware-aware model requires prior information about the underlying architecture of an execution system. It tights to the particular hardware, and hence needs to be rewritten to work optimally on a new hardware system. The hardware-independent model is a more flexible model based on functional decomposition. It depends only on the algorithm and hence it is more scalable. Also, the hardware-independent model can adapt to changes of the underlying hardware, and hence is easier to maintain. As illustrated, a hardware-independent model can run optimally on both single-DMA and dual-DMA systems.

The hardware-aware model uses less memory than the hardware-independent model, thus it might be the method of choice on systems with limited memory capacity, such as embedded systems. In addition, it can be used as a fall back solution to extend the system processing capability.

5 Streaming Models Stage Re-ordering

The aforementioned approaches are simple and theoretically optimal. They are straightforward to transfer from single-image processing to image-set processing through the generalization of basic image-set operators. However, the optimal performance is hardly achieved in practice. As we show here, the primary reason is the streaming degeneracy.

5.1 Forced Synchronizations

There are three primary reasons that performance degeneracies appear in streaming models:

- Synchronous function calls;
- Asynchronous stream mismatches;
- Cross-stream function calls.

The most common reason for an unintended synchronous function call is that the application requires an external call to a library function that was designed for synchronous execution. Another reason is the mixed use of synchronous and asynchronous functions. Our solution to the problem is to use a framework that fully supports asynchronous execution [40] for all essential functions. In addition to minimizing the demands for external function calls, it improves the uniformity and maintainability of the code.

Even when all functions support asynchronous execution, they might be designed using different strategies, which are often incompatible and can't work together efficiently. For example, a kernel function defined to run on a logical stream is incapable of running in-parallel with a data-transfer function on the physical stream with the same identity. These functions frequently require explicit synchronization to switch between the different asynchronous modes. Again, an unified framework prevents the misaligned execution models from happening. We have proved our asynchronous execution mechanism approach to be simple and effective. Supporting asynchronous execution at the run-time development level guarantees this uniformity, and therefore a cross-library execution model is achievable.

The third reason, cross-stream synchronization, occurs when a stream requires data from different streams. An example is the traditional implementation of reduction functions (sum, average, maximum/minimum values etc.) in CUDA. Though the computation runs on GPUs, outputs of these functions are copied from GPU memory to CPU memory to be used as parameters of subsequent calls or branching on CPUs. This is a cross-stream function between the computational stream on GPUs and the transfer data stream between GPUs and CPUs. Though the amount of data transfer between CPUs and GPUs is minimal, it requires any previous data transfers to be completed, flushing the pipeline and resulting in a wait. Figure 7 shows that the copy of the reduction result to the host, which takes only a negligible amount of time and has to be delayed until the data transfer pipeline becomes available. Meanwhile, the GPUs are idle, thus wasting their computational power.

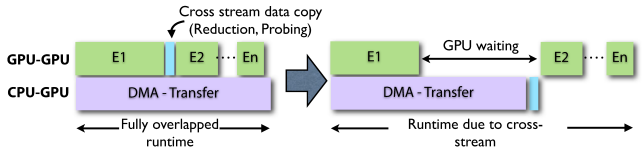


Fig. 7. Though it requires a minimum amount of data transfer, a cross stream function creates a forced synchronization that causes an unintended delay in the execution of the pipeline. In this example the GPUs are waiting for the reduction result to update on the CPU.

The popularity of reduction functions is the main obstacle for applying asynchronous models on existing GPU architectures. This cross-stream synchronization is difficult in profiling and might fail in some cases. Typically, the profiling tool, a probe, is inserted into the execution stream to collect data. As an event is triggered, the probing data needs to be copied from the device to the host for in-time visualization purposes. This process "steals" the data transfer pipeline, and unintentionally becomes a synchronization point. As a result, the profile result fails to measure the behaviour of the program in practice. This explains why general profiling tools such as the CUDA profiler [13] incorrectly reports the performance of streaming applications. The lack of profiling tools limits the understanding of streaming and pipeline processing techniques. Fortunately, the cross stream synchronization can be handled for most cases in practice. We propose two approaches for these problems.

5.2 Solutions for cross-stream synchronization

Our first solution to the reduction-like function is an on-device processing model that outputs the result only to device memory. However, this solution requires subsequent functions to use on-device parameters. While on-device parameters seem to add difficulties and incur a performance penalty on the program, we have found a solution that minimizes this influence based on texture caching. As shown on Figure 8, function calls using parameter caching are as fast as regular GPU function calls that allocate parameters from shared memory. The CPU code branching can be delayed or removed completely by moving it from CPUs to GPUs. Note that GPUs still provide maximum performance if the branching happens on the warp boundary (or SIMD width boundary). The same mechanism can be applied for profiling problems at the price of delayed visualizations. We collect the data in the device memory, and only release results to the host when an explicit synchronization is called, which is regularly required in a program. The delayed visualization, which allows for minimal invasive profiling, is the method of choice for most applications.

A more complete solution for the problem depends on a special mechanism on the platform that allows a non-invasive, small data transfers to instantly copy the data from and to the device without using the

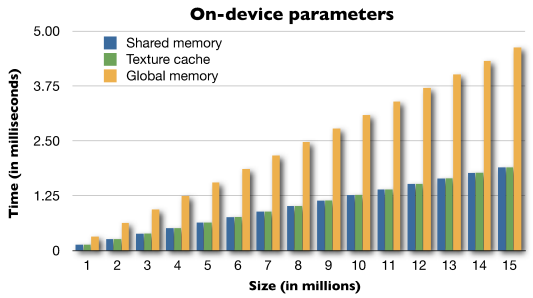


Fig. 8. Performance test with the GPU function. The texture memory access gives the same performance as the shared memory parameter models, global memory access is significantly slower.

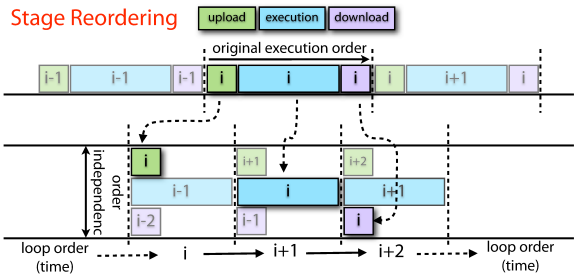


Fig. 9. The transformation from a synchronous model to an explicit streaming model preserves semantic correctness.

large data communication specialized DMA. This special communication channel is able to run concurrently in run-time with other CPU and GPU computations, and other data channels. This will naturally remove the need of in-device parameter functions, reducing the complication and improving the consistency of application code. It also allows in-time visualization for profiling tools, especially critical to real-time applications.

5.3 Re-ordering Pipeline Stages

In many cases a forced synchronization is still unavoidable, but its negative effects can be minimized using a reordering technique. This out-of-order execution is applied in modern compilers to reduce the number of mis-predicted branches. It helps to avoid data spilling, to keep instruction pipelines filled, and especially, to allow parallel execution on a system of multi-processors.

In the case of degeneracies in streaming modes, the reordering optimization cannot be done automatically using the compiler. The reason is that the upload and download tasks are I/O processes that have side effects. This constrains the order of function execution, and requires the compiler-generated code to execute in the same order as it appears in the API levels. Even worse, the forced synchronous functions impose a restriction in the order of the outputs. Therefore, reordering without compiler support must be done explicitly.

Allowing different streams to work in independent images gives explicit models a way to break the order-execution dependency inside the loop, thus replacing it

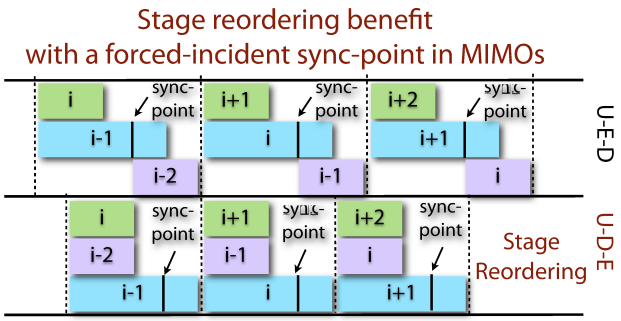


Fig. 10. Streaming optimization using a reorde ing technique eliminates the negative effect of force synchronous function.

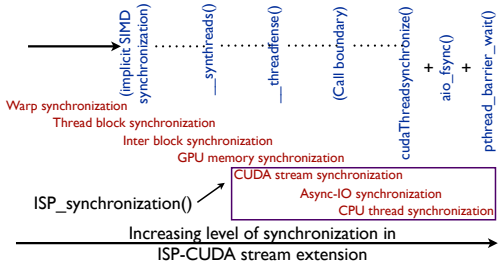


Fig. 11. Synchronization strategy for ISP_CUDA streaming extension model

with an equivalent order-independent streaming model. As shown in Figure 9, the order dependency of the original loop is still preserved in the order of loop execution. In other words, the logical correctness of the processing model is guaranteed by construction.

As the order of streams inside a loop loses importance, we can change the order of streams at the API level from the regular order of upload-process-download to upload-download-process, or even to process-upload-download. The ability to change the ordering allows streaming optimization, which is particularly effective when asynchronous stream degeneracies are unavoidable.

In the implicit model, when the synchronizations exists in the execution process, it is not possible to overlap the upload and download streams. The upload has to finish before the synchronization points, and the download only happens after the synchronization points. As shown on Figure 10, changing the order of streams in the code using the explicit model allows the upload and download streams to fully overlap, even when a synchronization point is present. Thus, reordering helps reduce the run-time per iteration, as well as the overall run-time. The ability to semantically reorder the stream execution in the code allows us to adapt a performance heuristic that profiles the performance and selects the optimal order.

6 Mapping Streaming Modes to Systems

It is critical in practice to map the streaming mode to the underlying functions of the system. In CUDA, streams have a one-to-one mapping to CUDA streams [14]. However, this relationship is not necessary true in other heterogeneous platforms. Fortunately, as CPUs have been used to coordinate operations in the system, we can exploit CPU threading models to define streams. In this model, each stream is controlled by a single thread with a fixed function. These function threads are created at the beginning of the program and destroyed at the exit. In the hardware-independent model, the number of threads is based on the number of stream functions, rather than the number of execution hardware in the system. The stream synchronization is performed using barriers. When two streams access the same resource, the resource is protected with a mutex to serialize the access. The job responsible for submitting streams is queued to be executed in order when resources become available.

This streaming model can also be exploited on the CPU with CUDA stream contexts. In the CUDA runtime model, streams are defined explicitly for GPU-runtime functions (through *cudaStream_t* objects), with no streaming mechanism for CPU counterparts. We use the CPU threading model to create multiple CPU execution streams. The synchronous barrier uses the GPU synchronization mechanism (e.g. *cudaThreadSynchronization*) and the CPU thread barriers (Figure 11).

Another option for asynchronous threading is to use the asynchronous data transfer (*AIO*) provided by Linux systems [33], or the equivalent *IO Completion Port* on Windows [17]. This is an option if the CPU execution streams only involve data transmission between external storage and the main CPU memory. The synchronization barrier is combined between the GPU synchronization and the asynchronous transfer barrier (Figure 11).

7 Asynchronous Processing Strategy

The generality of our asynchronous processing model allows us different implementation strategies to improve the performance of image-set processing functions. Data transfers are often the performance bottleneck for parallel systems, and have direct impact in the performance of asynchronous processing since devices have to wait for the data to become available. As discussed in Section 4.2, the system yields the maximum performance when the load among devices is balanced. We can improve the overall performance if we are able to increase computation on the processing devices.

This can be achieved using data compression. In this approach, data is stored in a compressed format to reduce the overall bandwidth. Compression algorithms are chosen based on the requirements of each application, which take into account the trade-off between the compression ratio and the data decompression quality. Data is received in the processing device, which decompress data to a format that can be processed. This decompression step increases the on-device processing

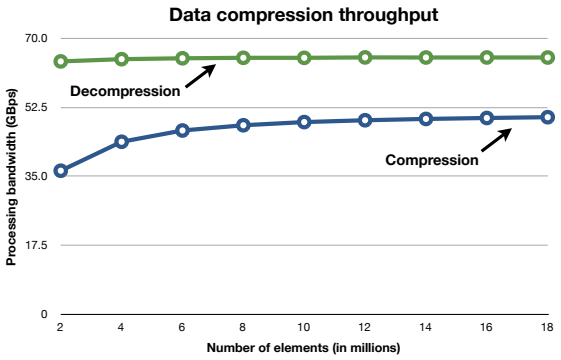


Fig. 12. Throughput measurement of our pseudo loss-less data compression for floating point inputs.

time and potentially impacts the load balancing. We might also consider to apply data compression to the generated output. This can reduce the data downloading bandwidth, as well as increase the processing load on the parallel processing devices, which might improve the overall load-balancing.

7.1 High performance pseudo loss-less data compression for floating-point data

Choosing the right data compression strategy for an out-of-core processing is essential. In this situation, the compression method must allow real-time processing, with compression and decompression rates that match the bandwidth of out-of-core devices. As data compression increases the overall processing time, it might have a negative impact if it becomes a bottleneck. Data compression implementations often requires a trade-off between compression ratio, data reconstruction quality and performance penalty. Depending on the type of data, it is possible to design simple but effective solutions. Here we consider floating point data, which is the most popular data format in image processing.

Floating-point data has irregular, dynamic representations which often lead to low compression ratios using general integer, dictionary-based techniques such as LZW, Gzip, Bzip [3]. These techniques are also developed specially for serial processing model, and make it difficult to implement effectively on parallel heterogeneous devices such as GPUs. Furthermore, data compression and decompression generally show data dependencies, which allow it to leverage parallel computation. We propose a simple pseudo loss-less compression [10] strategy, which leads to a simple compression and decompression strategy for floating-point data.

Floating-point operations are prone to inaccuracy. It is generally safe to assume that linear operations on floating-point data do not introduce arithmetical errors to existing algorithms. There is an interesting observation regarding the floating-point presentation of numbers in the range of $[2n; 2n + 1)$, which share the same leading exponential and sign bits. These bits are considered redundant for data storage, which allows a reduction in the number of information bits from 32-bits to 24-bits. For this purpose, fractional data is represented using a normalized linear mapping from [a; b] to the [0.5; 1) range. This mapping yields an immediate 25% compression ratio, as we can store four floating-point numbers using only three 32-bit integers. The only extra information required for the mapping process is the range of the input data that can be efficiently computed using a parallel reduction algorithm. This range is also needed to restore the compressed data to the initial value in the decompression process. Both the reduction and mapping operator use the highest memory bandwidth available, equivalent to a memory copy [27]. Figure 12 shows that this strategy can run at 50 GBps for compression and at 65 GBps for decompression.

An additional advantage of using our compression scheme is the ability to immediately extend the processing capability of each level in our out-of-core processing hierarchy by a factor of 1.33. Therefore, if we are capable of processing 300 subjects on the first level, the compression stream is capable of processing 400 subjects without requiring significant changes to existing algorithms.

In addition, the linear mapping preserves the data coherency and allow it to be exploited by further compression schemes, such as differential compression techniques. The loss-less compression predictive coding of Isenburg *et al.* [32] or FPC coder by Burtscher and Ratanaworabhan [8] can also be applied, as these coding techniques can be implemented efficiently on heterogeneous systems. Note that our compression scheme requires reduction, which potentially leads to cross-stream synchronization. This can be prevented using prior input information or our on-device parameter models.

8 Hardware-independent model extensions

8.1 Extension to a Full Out-of-Core Framework

The extension from a partial out-of-core model (with one level of memory hierarchy) to a *full* out-of-core model (with two-memory levels) comes naturally with the hardware-independent model. We realize the transition to a fully out-of-core model by adding two more stages to the algorithm pipeline. The first stage is at the start of the pipeline, corresponding to the upload from disk to CPU memory, and the second stage is at the end of the pipeline, corresponding to the download from the CPU memory to disk. The execution of this model for MIMO operation is given in Figure 13.

Using the same logic as in the partial out-of-core model, we can prove that the hardware-independent model for out-of-core processing is optimal. We use the term *full* to indicate that data can be stored on disk of a single machine. Moreover, the hardware-independent model can be further extended to other out-of-core models, such as the one with a data stream on the network in a system with higher memory hierarchy levels, and we can still prove that the proposed models are optimal.

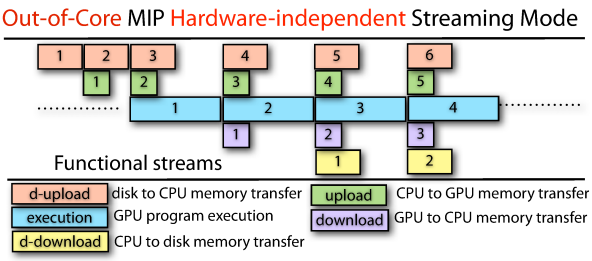


Fig. 13. The implementation of hardware-independent model for "full" out-of-core image-set processing.

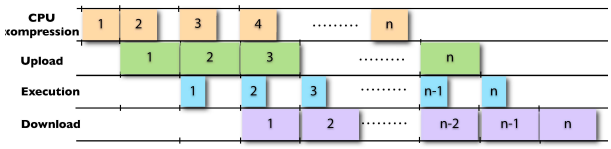


Fig. 14. Extension of the hardware-independent model with more CPU processing stages. The architecture is another option to improve the load balancing for heterogeneous systems.

8.2 Extension with more CPU processing stages

In the discussion above, CPUs are used as control devices that perform external IOs and coordinate streaming units. There are circumstances when the processing time is higher than data transfers. In such cases, we can improve load balancing using the CPU (or multi-CPUs) as processing devices to reduce the workload of GPUs.

Due to the lower processing capability, CPUs are suitable for pre-processing stages (e.g. raw data processing, data normalization, data compression, etc.) or post-processing stages (e.g. data de-normalization, data decryption, etc.). Figure 14 shows an independent asynchronous processing model with input data compression on CPUs. The strategy increases the effective bandwidth between CPUs and GPUs without adding further workload to GPU processing devices. This is also the model for a hybrid processing system, which aims to exploit the computational power of the underlying hardware.

9 Results

The system we used in our experiments is a PC desktop, Intel Core i7-980X, 12-GB DDR3 1600, with a single NVIDIA GTX 480. Communication from the host to GPU is via the external x16 PCIe bus and is controlled by a single DMA. The program is compiled with CUDA NVCC 3.2. Run-time of each function is measured in milliseconds.

We made a synthetic test on a data set of 32 volumes, sized $256\times256\times256$. The test mimics a typical out-of-core image-set processing program using three processes: upload, execution, and download. Note that the execution time and data-transfer times scale propotionally to the number of images and the sizes of the image, we

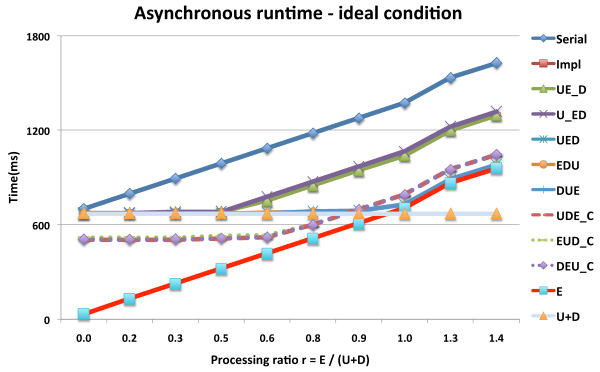


Fig. 15. Runtime comparison of different streaming strategies in ideal conditions.

also achieve similar performance curves with different number of images ranging from 10 to 180 (the maximum number of volumes we can fit onto the 12GB of memory).

As mentioned in Section 7.1, floating point compression strategy allows us to extend the processing capability in the first level to 240 compressed volumes immediately (a higher number might be achieved with more sophisicated compression schemes).

The existing architecture on commodity hardware has a single DMA unit, therefore the upload and download processes have to be performed sequentially. This information allows a two-device, hardware-aware model with only two memory buffers. There are two options for its implementation: (1) the upload of the k-th volume in parallel with the execution and the download of $(k-1)$-th volume (U_ED);(2) the upload and execution of the k-th volume in parallel with the download of $(k-1)$-th volume (UE_D). where U, E, and D stand for Upload, Execution and Download respectively. Our hardware-independent model still decomposes the algorithm into three processes regardless of the system configuration. There are six permutations for the implementation of the hardware independent model: UED, UDE, EDU, EUD, DUE, and DEU. We have also experimented with six permutations of our compression scheme, displayed in dashed lines and the post-fix C in the name (i.e. UED_C). We also keep track the best performance among non-compression schemes—the HI curve, and among compression schemes—the HI_C curve.

Since it is not possible to perform data upload and download in parallel, there are actually only 3 distinct performance pairs UED-DEU, UDE-DUE, and EDU-EUD, thus we show only one performance curve for each pair. The same approach is applied for the compression schemes.

9.1 Full asynchronous processing

First, we tested using the ideal case: a full asynchronous processing function without a single synchronous call in the execution. We measured the influence of the ratio between computation and data transfer (processing ratio) on the performance of different asynchronous processing models, denoted by $r_e = E/(U+D)$. This

ratio indicates different types of out-of-core functions: data-transfer dominance ($r << 1$), processing dominance ($r >> 1$), and balanced functions ($r \approx 1$). In the ideal case, the results on Figure 15 show:

- In all tests, the six hardware independent implementations give us the same performance. The six compression schemes also give identical results. The hardware-aware and implicit models give similar runtimes;
- If the function is transfer-dominant ($r_e < 0.5$), all models achieve the optimal solution (equivalent to the data transfer), and the execution time is completely hidden. The compression scheme which mainly reduces transfer time gives an immediate speed up of 25% over non-compression technique;
- When the execution time is larger than the upload or downloading time, the first two models still give strong performances, approximately $\mathcal{T}_u + \mathcal{T}_e$. However, it is not the optimal of $max(\mathcal{T}_u + \mathcal{T}_d, \mathcal{T}_e)$ achieved with the hardware-independent model. In this case, the hardware-independent model is faster than the hardware-aware model because the awareness from the hardware system requires that a double-image memory buffer is used instead of a triple one used by a hardware independent model. In this configuration, it is impossible for the hardware-aware models to have a single stream with both the upload and the download when the other stream is only processing. This condition is required to achieve the best performance;
- When the function is balanced or processing-dominant ($r_e \geq 1$), the hardware-independent model gives the optimal runtime $\mathcal{T}_e$ and the data transfer is completely hidden. Note that this is also the condition for ISP out-of-core functions to outperform ISP in-core implementation since the in-core version will spend $\mathcal{T}_e + n \times T_u$;
- The asynchronous function gives the best speedup in comparison to the synchronous models when the loads between two execution units are balanced ($r_e = 1$);
- The turning point of the compression scheme happens earlier than non-compression counterparts due to less amount of data to transfer and the adding compression/decompression load to execution process;
- When the process is balanced or execution dominant, the execution fully hides the data transfer. The compression scheme is no-longer effective, and might even reduce the overall performance.

9.2 Synchronous functions

Second, we tested a synchronous function. We fixed the run-time of three basic processes, but changed the position of the synchronous function inside the execution process to measure the influence of sync points inside the functions to different streaming models through the synchronous ratio $r_s = E1/(E1 + E2)$. From our experiments (detailed result graphs can be found in the appendix) we conclude:

- The position of the sync point within the asynchronous code directly affects the performance of the given implementations;
- The implicit model no longer gives us the optimal result and is as slow as the synchronous implementation. It simply cannot find a schedule for asynchronous execution;
- The hardware-aware model cannot give optimal results in all the tests. However, it is still far better than the implicit model. Note that their two implementations also give different runtimes;
- The three hardware-independent pairs give different performance characteristics but in essence, the best result is always achieved with one of the hardware-independent implementations;
- When the execution is low, the UED (or DEU) strategy gives the best performance. Similarly, the UED_C (or DEU_C) is the best among compression schemes;
- As the execution increases, the EDU (or EUD) performs better in the low synchronous ratio region, the UED (or DEU) is favourable in the middle range, while the DUE (or UDE) scheme is best in the upper range. The same arguments hold for the compression schemes;
- Similar to the ideal condition, a compression scheme shows the effectiveness when the execution ratio r_e is low, but it becomes less efficient when this ratio increases and is even slower than non-compression approaches when this ratio is high (above the balance region).

9.3 Regular out-of-core functions

On the third experiment, we focus on the regular out-of-core function sets such as a maximum value of all images, normalization, averaging, Gaussian filtering, product (energy computation), temporal image smoothing using bilateral filter for movie data, and atlas building. The results from Table 1 confirm that when the computation only requires simple functions (max, product, normalization, averaging, etc.), the asynchronous streaming does give you the benefit of hiding the computational cost. However, it is negligible in comparison to the transfer cost. The compression scheme significantly improves the performance due to reduced amounts of data. As the complexity of the functions increases (for example, Gaussian filtering functions), we start seeing significant benefits of asynchronous streaming strategies, especially with the hardware-independent model. On the other hand, for expensive processing functions such as bilateral temporal image filtering for movie data, we can completely hide the data transfer time by execution; however, it also leads to low performance benefits in comparison to a simple synchronous approach.

In atlas construction, which is taken on the ADNI dataset that we mentioned on Figure 1, as we increase the complexity of computational functions and reduce the cost of data transfer by merging all the functions together on a single loop, we yield significant performance improvement over the synchronous out-of-core version. The performance is comparable to the incore performance (execution time only) while we could process a significant amount of data much larger than that of an in-core version. Besides increasing the performance about

10%, the compression scheme allows us to process 240 brain volumes on the first out-of-core level in comparison to 180 subjects with a non-compression approach.

Overall, results confirm our theoretical analysis. All the strategies are able to achieve optimal performance; however, only the hardware-independent model gives the best performance in all tests. In the degenerate cases, the implicit model completely fails. The presence of synchronization points makes it impossible to find a efficient scheduling automatically. Note that in this case, a greedy approach, which immediately executes whenever the resource is available, also fails. The hardware-aware model gives better performance even with the degenerate cases, although it is not optimal. It is always possible to find the best runtime between hardware-independent implementations. In other words, the optimal performance is always achievable with the hardware-independent model. Our compression scheme could significantly increase the performance from 10 to 25% with minimal additions to the existing framework.

10 Conclusions

In this paper, we presented an optimized, parallel, image-set processing framework on heterogeneous commodity systems extending from the existing single-image, parallel processing framework. We introduced image-set operators, serving as the connection between the single-image processing model and the image-set processing variant. We proposed the MIMO and MISO image-set operators, which are used to construct other image-set operators, allowing us to build a image-set processing framework.

Optimal streaming models were presented for the image-set processing framework. We analyzed the advantages and disadvantages of various streaming strategies, and proposed a generalized streaming model based on functional decomposition that is optimal, hardware-independent, and highly scalable on future hardware. Experimental results show that our hardware-independent model adapts to underlying hardware configurations, out-performs other streaming strategies, and gives optimal performance in all tests.

We also evaluated the efficiency of streaming models, and presented a quantitative evaluation that serves as a model for developers. We investigated an optimal streaming strategy in unfavorable conditions based on reordering from order-independent properties of the explicit-streaming models. We also gave insights to the causes of unfavorable streaming conditions that help developers locate the performance degradation points in their implementations. Though we use a GPU computational model to illustrate the efficiency, our framework makes no specific assumptions about the underlying architecture and hence can be generalized to any heterogeneous parallel processing system.

Function	U	E	D	Sync	Impl	H_A	H_I	HI_C
Max	347	13	0	360	349	349	349	280
Energy	692	20	0	710	698	700	698	540
Averaging	347	20	11	378	360	363	361	293
Normalization	347	28	322	694	696	687	677	567
Gaussian	347	431	322	1099	735	770	678	613
Bilateral	448	7342	418	8205	NA	7832	7385	NA
Atlas	201446	213423	135958	555204	NA	372567	340356	292567

TABLE 1
Runtime comparison of regular functions in practices with different streaming strategies.

Acknowledgments

We would like to thank Thomas Fogal and the anonymous reviewers for insightful discussions and constructive comments that helped us to substantially improve this paper. This research has been funded by NIH grant: NIBIB 5R01EB007688, UCSF grant NCRRR (P41 RR023953), CNPq 200498/2010-0, 569239/2008-7, and 491034/2008-3), and NSF grants CNS-0751152 and 0844572 as well as the Intel Visual Computing Institute. Linh Ha is partially supported by Vietnamese Education Foundation fellowship.

References

[1] 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.

[2] J. Bittner, M. Wimmer, H. Piringer, and W. Purgathofer, *Coherent hierarchical culling: Hardware occlusion queries made useful.*, Comput. Graph. Forum (2004), 615–624.

[3] G.E. Blelloch, *Introduction to data compression*, Carnegie Mellon University (2010).

[4] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny, *A model and compilation strategy for out-of-core data parallel programs*, SIGPLAN **30** (1995), no. 8, 1–10.

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

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

[7] A.D. Brown, T.C. Mowry, and O. Krieger, *Compiler-based i/o prefetching for out-of-core applications*, TCS **19** (2001), no. 2, 170.

[8] M. Burtscher and P. Ratanaworabhan, *High throughput compression of double-precision floating-point data*, DCC'07, Proceedings, 2007, pp. 293–302.

[9] E. Caron, F. Desprez, and F. Suter, *Out-of-core and pipeline techniques for wavefront algorithms*, IPDPS' 05. Proceedings. **01** (2005).

[10] T.-J. Chen and K.-S. Chuang, *A pseudo lossless image compression method*, CISP' 10, vol. 2, Oct 2010, pp. 610 –615.

[11] Y.J. Chiang, J. El-Sana, P. Lindstrom, R. Pajarola, and C.T. Silva, *Out-of-core algorithms for scientific visualization and computer graphics*, IEEE Vis. (2003).

[12] G.E. Christensen, M.I. Miller, M.W. Vannier, and U. Grenander, *Individualizing neuroanatomical atlases using a massively parallel computer*, Computer, vol. 29, 1996, pp. 32–38.

[13] NVIDIA Corp, *Compute Visual Profiler User Guide*, Oct 2010.

[14] ———, *NVIDA CUDA Programming Guide 3.2*, Oct 2010.

[15] B.C. Davis, P.T. Fletcher, E. Bullitt, and S. Joshi, *Population shape regression from random design data*, IJCV **90** (2010), no. 1, 255–266.

[16] E. Derzapf, N. Menzel, and M. Guthe, *Parallel view-dependent out-of-core progressive meshes.*, VMV, 2010, pp. 25–32.

[17] T.R. Dial, *Multithreaded asynchronous io & io completion ports*.

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

[19] A. Eklund, M. Andersson, and H. Knutsson, *Phase based volume registration using CUDA*, ICASSP' 10, Mar 2010, pp. 658 –661.

[20] T. Engelhardt and C. Dachsbacher, *Granular visibility queries on the gpu*, I3D '09, Proceedings, I3D '09, 2009, pp. 161–167.

[21] R. Farias and C.T. Silva, *Out-of-core rendering of large, unstructured grids*, ICGA' 01 **21** (2001), 42–50.

[22] P.T. Fletcher, R.T. Whitaker, R. Tao, M.B. DuBray, A. Froehlich, C. Ravichandran, A.L. Alexander, E.D. Bigler, N. Lange, and J.E. Lainhart, *Microstructural connectivity of the arcuate fasciculus in adolescents with high-functioning autism*, NeuroImage **51** (2010), no. 3, 1117–1125.

[23] M.J. Flynn, *Some computer organizations and their effectiveness*, ITC' 72 **C-21** (1972), 948 –960.

[24] 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.

[25] L.K. Ha, J. Kruger, P.T Fletcher, S. Joshi, and C.T. Silva, *Fast parallel unbiased diffeomorphic atlas construction on multi-graphics processing units*, EGPGV' 09. Proceedings., 2009.

[26] L.K. Ha, J. Krüger, S. Joshi, and C.T. Silva, *Multiscale Unbiased Diffeomorphic Atlas Construction on Multi-GPUs*, vol. I, Elsevier, Jan 2011.

[27] M. Harris, *Optimizing Parallel Reduction in CUDA*, 2007.

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

[29] H. Hoppe, *Progressive meshes*, SIGGRAPH' 96, Proceedings, 1996, pp. 99–108.

[30] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, and D. Manocha, *Memory-scalable gpu spatial hierarchy construction*, TVCG **17** (2011), no. 4, 466 –474.

[31] C. Hu, G. Yao, J. Wang, and J. Li, *Transforming the adaptive irregular out-of-core applications for hiding communication and disk i/o*, OTM'07. Proceedings. **Part II** (2007).

[32] M. Isenburg, P. Lindstrom, and J. Snoeyink, *Lossless compression of predicted floating-point geometry*, Comput. Aided Des. **37** (2005), 869–877.

[33] M.T. Jones, *Boost application performance using asynchronous i/o*, IBM developerWorks (2006).

[34] S. Joshi, B. Davis, M. Jomier, and G. Gerig, *Unbiased diffeomorphic atlas construction for computational anatomy*, NI **23** (2004), 151–160.

[35] L. Lamport, *How to make a multiprocessor computer that correctly executes multiprocess programs*, ITC' 79 **28** (1979), 690–691.

[36] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, *Fast bvh construction on gpus*, EG' 09, vol. 28, 2009, pp. 375–384.

[37] A. Macovksi, *Tolerating latency through software-controlled data prefetching*, en.scientificcommons.org (1994).

[38] H. Meuer, *China grabs supercomputing leadership spot in latest ranking of world's top 500 supercomputers*, Nov 2010.

[39] T.C Mowry, A.K. Demke, and O. Krieger, *Automatic compiler-inserted i/o prefetching for out-of-core applications*, OSDI' 96, Proceedings **30** (1996), no. si, 3–17.

[40] 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.

[41] 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.

[42] M. Roberts, M.C. Sousa, and J.R. Mitchell, *A work-efficient gpu algorithm for level set segmentation*, SIGGRAPH' 10, Posters, 2010, pp. 53:1–53:1.

[43] D. Scherzer, L. Yang, and O. Mattausch, *Exploiting temporal coherence in real-time rendering*, SIGGRAPH Asia' 10 Courses, SA '10, 2010, pp. 24:1–24:26.

[44] N. Snavely, R. Garg, S.M. Seitz, and R. Szeliski, *Finding paths through the world's photos*, T0G **27** (2008), 15:1–15:11.

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

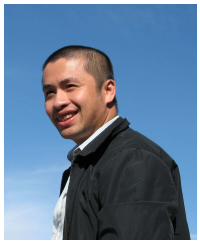
[46] N. Sundaram, A. Raghunathan, and S.T. Chakradhar, *A framework for efficient and scalable execution of domain-specific templates on gpus*, IPDPS '09 **0** (2009), 1–12.

[47] D. Womble, D. Greenberg, R. Riesen, and S. Wheat, *Out of core, out of mind: Practical parallel i/o*, SPLC' 93. Proceedings. (2002), 10–16.

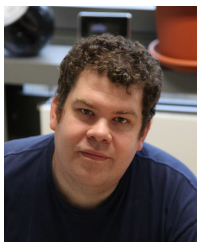
[48] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha, *Cache-oblivious mesh layouts*, TOG **24** (2005), 886–893.

[49] S.-E. Yoon, B. Salomon, R. Gayle, and D. Manocha, *Quick-vdr: out-of-core view-dependent rendering of gigantic models*, TVCG **11** (2005), no. 4, 369 –382.

[50] K. Zhou, Q. Hou, R. Wang, and B. Guo, *Real-time kd-tree construction on graphics hardware*, TOG **27** (2008), 126:1–126:11.



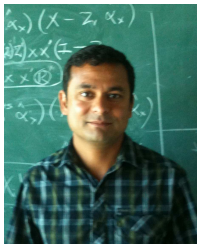
Linh K. Ha studied Electronics and Telecommunications at UTH, Vietnam, 2003. He finished ME from College of Technology, VNUH, 2005. He received the VEF Fellowship for young outstanding Vietnamese scholars to continue graduate studies in the US and was admitted to University of Utah, Computer Science Department, 2005. He joined Visualization and Geometric Computing (VGC) group at Scientific Computing Institute (SCI) under the supervision of Prof. Claudio Silva. He finished PhD in Aug 2011.



Jens Krüger received his PhD in Computer Science from the Technische Universität München and shortly after joined the Scientific Computing and Imaging (SCI) Institute. He is currently working at the the Cluster of Excellence in 2009 to head the Interactive Visualization and Data Analysis group. In addition to his position within the Cluster, Jens also holds an adjunct faculty title of the university of Utah and is a principal investigator of multiple projects in the Intel Visual Computing Institute.

João L. D. Comba received a PhD in Computer Science from Stanford University under the supervision of Leonidas J. Guibas, a MSc degree in Computer Science from the Federal University of Rio de Janeiro (UFRJ), Brazil, and a BSc in Computer Science was given by the Federal University of Rio Grande do Sul, Brazil. Dr Comba is currently an Associate Professor in the Graphics Group at the "Instituto de Informática" of the Federal University at Rio Grande do Sul (UFRGS), Brazil. His main research interests are in visualization, computer graphics, spatial data structures, graphics hardware and HPC.

Cláudio T. Silva is Professor of Computer Science at NYU's Polytechnic Institute. He received his Ph.D. in computer science at SUNY-Stony Brook in 1996. He coauthored more than 175 papers and eight U.S. patents. He is currently in the editorial board of Computing in Science and Engineering, Computer and Graphics, The Visual Computer, and Graphical Models. He was general co-chair of IEEE Visualization 2010, and papers co-chair of VIS 2005 and 2006. He received IBM Faculty Awards in 2005, 2006, and 2007, and best paper awards at IEEE Visualization 2007, IEEE Shape Modeling International 2008 and the 2010 Eurographics Educator Program. His work is funded by grants from the NSF, NIH, DOE, IBM, and ExxonMobil.



Sarang Joshi received his D.Sc. in Electrical Engineering from Washington University in St. Louis. After his D.Sc., Dr. Joshi was Director of Technology Development at IntellX, a Medical Imaging start-up company which was later acquired by Medtronic. He returned to academia as an Assistant Professor of Radiation Oncology and an Adjunct Assistant Professor of Computer Science at the University of North Carolina in Chapel Hill. Currently he is an Associate Professor in the Department of Bioengineering and a member of Scientific Computing and Imaging Institute at University of Utah. His research interests are in the field of Computational Anatomy and statistics in high dimensional nonlinear spaces with applications to Medical Image Analysis.