Eurographics Symposium on Parallel Graphics and Visualization (2011)
T. Kuhlen, R. Pajarola, and K. Zhou (Editors)

Optimal Multi-Image Processing Streaming Framework on Parallel Heterogeneous Systems

Linh K. Ha, Jens Krüger, Joao Comba, Sarang Joshi, Cláudio T. Silva

Scientific Computing and Imaging Institute, University of Utah

Abstract

Atlas construction is an important technique in medical image analysis that plays a central role in understanding the variability of brain anatomy. The construction often requires applying image processing operations to multiple images (often hundreds of volumetric datasets), which is challenging in computational power as well as memory requirements. In this paper we introduce MIP, a Multi-Image Processing streaming framework to harness the processing power of heterogeneous CPU/GPU systems. In MIP we introduce specially designed streaming algorithms and data structures that provides an optimal solution for out-of-core multi-image processing problems both in terms of memory usage and computational efficiency. MIP 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 MIP out-of-core solution could achieve the same performance as the in-core solution. We demonstrate the efficiency of the MIP framework on synthetic and real datasets.

1. Introduction

Multi-image processing is an advanced image processing technique that is widely used in medical imaging [CMVG96], video processing [BJW95, RDK*98], astronomy [Boy92, Ala92], visual robot control [DM95], virtual reality [SSS06, SGSS08], modeling and reconstruction [GSC*07, HE07], etc.

One example is the atlas construction technique, which plays a central role in medical image analysis, particularly in understanding the variability of brain anatomy [CMVG96, DFBJ07, JDJG04]. The atlas construction projects a large set of images to a common coordinate system, creates a statistical average model of the population, and performs a regression analysis of anatomical structures. This average serves as a deformable template which maps detailed atlas data such as structural, developmental, genetic, pathological, and functional information onto the individual or entire population of the brain.

Examples like this one show how multi-image processing techniques provide benefits over single-image processing, at the expense of introducing major computational challenges: First, they involve huge amounts of data that easily exceed the direct processing capability of the system. Second, they require massive amounts of computation, which results in the computations requiring days or even months to complete. As a result, using multi-image techniques often involve super-computing systems [CMVG96] or large-scale clusters to run [SGSS08, HKF*09], which limits the use of multi-image processing techniques to large laboratories. A solution based on commodity hardware will make this technique available to smaller labs, increase the influence of these techniques in research, and presents robust solutions for many existing problems.

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

- We introduce a high-performance, multi-image processing framework with a proof-of-concept optimal streaming model.
- We define basic building blocks of a general framework which allow efficient-implementation multi-image algorithms.
- We introduce concepts for implicit and explicit pipelining and prove that these are optimal solutions.
- We analyze reasons for streaming degeneracy and provide a solution based on an order-independent model.
- Our performance analyses serve as the guidance to help

DOI: 10.2312/EGPGV/EGPGV11/001-010

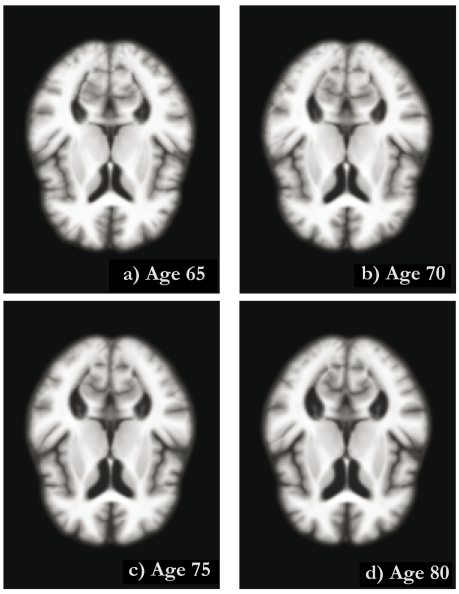

Figure 1: *Age regression anlysis on the ADNI dataset by computing the average brain atlases at different ages (65, 70, 75, and 80) confirms the proposition that fluid space is larger because brains atrophies overtime. This analysis, however, could only be performed if the system is capable of processing the whole dataset of 300-healthy brain-images* $(144 \times 192 \times 160)$

developers to profile performance and to make quantitative decisions.

2. Related work

In the last decade, there is an emerging trend in High Performance Computing research community (HPC) to use heterogeneous processing systems such as Cell processors, FPGAs, and multi-core GPUs to replace the conventional supercomputing model. Super-computing systems based on the heterogeneous model have been successfully exploited in some of the fastest computing system, such as the current number one super-computer Tianhe-1A, the first computing system to achieve 2.5 petaflop/s [Meu10].

With hundreds of simple, computation-centric processing cores, the GPU processing model has proven to be highly scalable to many problems especially image processing by providing massive computational power. Modern GPUs can offer a few Tera-flops of peak performance per unit, providing processing power equivalent to a super-computer in mid-90s, while being much 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 GPU centric processing model. There have been a number of image processing applications implemented on GPUs [RSM10, SRC09, EAK10], most of which achieve from 20x to several magnitudes of speedup over CPU versions. Conceptually, our streaming framework is an extension to the idea of the fast GPU image-processing framework by Ha *et al.* [HKF*09, HKJS11]. Their method achieved 60x speed up in comparison to an optimized, fully-parallel version running on an eight-core Xeon server for Greedy Iterative Differmorphic Atlas construction problem.

While the use of GPUs appears to be a good solution to the computing requirements of multi-image processing techniques, the large memory footprint remains an open problem. Though providing ample memory bandwidth, the size of the on board GPU memory is very limited. But as GPU programs can only access on-board, all required data needs to be present on the card, so out-of-core methods need to be employed.

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 [WGRW02]. The second approach is to use compiler directed I/O to convert a program from in-core to out-of-core [BCK*95, MDK96, BMK01]. For programs with complicated data dependencies this approach is not as effective as the third approach that we use here: the explicit I/O controls by developers. These methods concentrate on techniques to improve the cache coherency such as *caching* and *prefetching* [Mac94, CDS05, CESL*03, HYWL07, BWP04] to reduce the I/O necessary for blocks already in main memory and/or by overlapping I/O operations with main-memory computations. The methods exploit particular computational properties of each individual problem as part of the algorithm design. While the explicit I/O controls are mostly application-specific, our method is able to be applied to a wide class of applications such as the out-of-core multi-image processing.

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 the resource usage. It can also hide data transfer by prefetching data. When processing units finish current tasks, they can start the next tasks without delay. In many circumstances, using this model significantly increases the overall system throughput.The similar concept has been successfully applied in video processing pipeline where video encoding/decoding/composition and filtering inherently require out-of-core multi-image stream processing [BJW95, RDK*98].

The asynchronous processing is realized with streaming

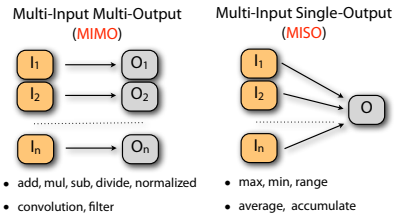

Figure 2: *Basic Multi-Image Operators*

models for both tasks and data. Streaming is an efficient model for parallel processing in that a task is divided into smaller entities to allow their 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. A stream in practice may be either data-based or tasked-based or even a mixture. The only restriction in a stream is the execution order that is satisfied by a sequential consistency model [Lam79], 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. Multi-Image Processing Operators

As we can see from an example of the atlas construction Algorithm 1 [HKJS11], a multi-image algorithm involves several multi-image operations, most of which are direct extensions of single-image processing operations through a for loop over all the input. We build our multi-image processing framework upon the single-image high-performance multi-scale processing framework proposed by Ha *et al.* [HKF*09] so that we are able to exploit the optimized performance of the existing framework.

We define the multi-image processing framework using a construction method that builds regular multi-image operators from basic building blocks. This strategy allows us a fine-grained and multi-level parallelism in that we could exploit different execution strategies on each implementation level to make use of available resources. Here, we classify basic multi-image operators into two main groups based on Flynn's taxonomy [Fly72]: the Multiple-Input-Multiple-Output operators (MIMO) and the Multiple-Input-Single-Output operators (MISO).

The basic MIMO operators are defined as functions with equal numbers of inputs and outputs, whereas the n-th output image depends solely on the n-th input image(Figure 2a). These functions are the most frequently used in multi-image processing as they are direct extensions of single-image operations. Examples for such operations include adding, shifting, scaling, smoothing, filtering, de-noising images, and normalizing the intensity range.

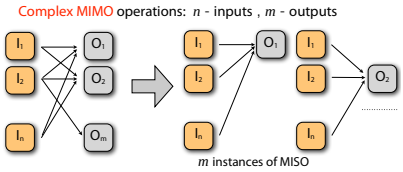

Figure 3: *General Multi-Input Multi-Output operators*

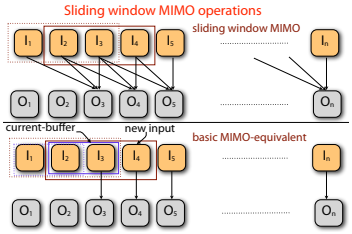

Figure 4: *Sliding window MIMO operators*

The MISO operators, as illustrated in Figure 2b, produce a single or few outputs. Examples for such operations include the computation of an average image, the image energy, cross-correlation, cross-product of images, and finding the maximal and minimal values.

The implementation of general multi-image operators is based on a decomposition strategy that breaks a complex function into multiple, basic operations. For example, a general MIMO function that has a number of outputs M which is different from the number of inputs N, and the k-th output depends on multiple inputs, could be implemented as M instances of a MISO operator as shown on Figure 3.

Another group of frequently used multi-image operators is the sliding-window operator (Figure 4a). 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 4b, 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 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 multi-image functions based on the basic MIMO and MISO functions. We focus our discussion on how to efficiently implement these out-of-core operators.

Note that the framework of Ha *et al.* already has support for multi-image and large data processing through the 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. Both approaches, however, have the limitation that they depend on the total amount of system memory. The out-of-core approach we introduce here, however, has no restrictions on data input and can

process the entire 3D-image brain dataset in a PC desktop equipped with commodity GPUs. Hence, our solution is more complete and accessible to researchers and scientists.

We also offer a more flexible solution to existing methods with two levels of streaming operations: out-of-core GPU 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, the dataset does not even fit into CPU memory and the data must be transferred through two memory levels: between disks and CPUs, and between CPUs and GPUs. 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. MIP Out-of-core Framework

We use a synchronous implementation of the MIMO (Algorithm 2) and MISO (Algorithm 3) operators as references for the correctness and performance improvement of our asynchronous implementations. We compare different methods to implement out-of-core multi-image operations: an implicit model, a hardware-aware model, and a hardware-independent model. We will prove that the proposed strategies are optimal. But first, lets do some analyses on the best achievable performance of an asynchronous algorithm.

4.1. Asynchronous optimal performance analyses

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, the other for downloading data). For performance analysis, we use following notation:

- n : the number of input images
- n_s : the number of execution units
- $\tau_{i,j}$: the runtime of the i-th execution unit on the j-th input image.
- T_s, T_a : the total synchronous/asynchronous processing time
- T_u, T_e, T_d: the uploading, executing, and downloading runtime per image (Figures 5, 6, 7).
- $\mathcal{T}_i$ the total amounts of time spent by the execution unit i
- $\mathcal{T}_u = n \times T_u, \mathcal{T}_e = n \times T_e, \mathcal{T}_d = n \times T_d$: the total amounts of time spent on upload, execution and download process.
- $\mathcal{T}_{max} = max(\mathcal{T}_1, \mathcal{T}_2, \cdots \mathcal{T}_{n_s})$ the maximum amounts of time spent by a single execution unit.

Our analysis is based on the assumption that all images have similar sizes, and therefore require almost the same amount of running time. This assumption is normally satisfied with pre-processing multi-image data.

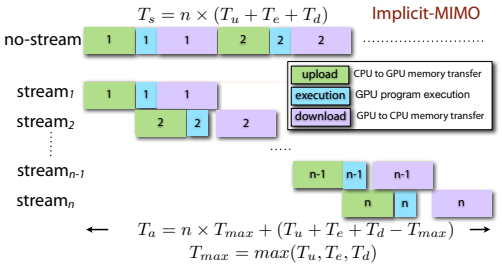

Figure 5: *Implicit processing model for MIMOs*

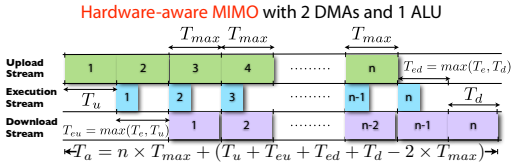

Figure 6: *Pipeline processing model for Multiple Images Multiple Output*

First, we determine the optimal asynchronous runtime, which we use as a reference to evaluate the efficiency of proposed implementation method. In the ideal case, all execution units run independently parallel. However, as a single execution entity, they perform tasks in sequential order. The total amounts of time that an execution unit spends is $\mathcal{T}_i = \sum_{j=1}^{n} \tau_{i,j}$ that equals $n \times \tau_i$ where τ_i runtime of i-th stream on a single-image. Since the multi-image operation is only completed when all the execution units have completed their tasks, the runtime the entire operation will be 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 runtime that the system can accomplish. Note that with the hardware configuration of upload, execution, and download units $\tau_{max} = T_{max} = max(T_u, T_e, T_d)$ (Figure 5).

4.2. Implicit Streaming Model

The *implicit streaming model* (Algorithms 4) is solely based on data parallelism that assigns each image to a stream which works as a logical execution unit that performs the entire processing pipeline (Figure 5). As streams operate on different memory spaces, the data transfer on a stream can be overlapped with processing tasks on other streams. This is a contrast to the *explicit streams* (Algorithms 5 6): *hardware-aware* and *hardware-independent* models, 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 5 illustrates the execution of an implicit streaming model for a MIMO problem (Algorithm 4). It can be seen that with the number of images being significantly larger than the number of streams, the overall processing time is

approximately $n \times T_{max}$ which is the optimal runtime of asynchronous processing.

4.3. Hardware-Aware Streaming Model

The execution of the hardware-aware processing model for MIMO problems is illustrated in Figure 6. In this model, there are three streams mapping to three execution devices. 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 prior information about the architecture of the underlying system. That is, it requires different implementation on different hardware.

4.4. 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 actual hardware configuration, we assume that there exists one special execution unit for every task, and we can assign each task a single stream. In the case of MIMO operations, there are three primary tasks to apply to each image: the data upload, the processing, and the data download. On a system with two data transfer units and one processing unit, it results in a streaming scheme similar to hardware-aware models; consequently, this model also achieves the optimal runtime.

Normally, however, there are more tasks than the actual number of execution units. In this case it is possible that several tasks are mapped to the same execution unit, for example, data uploading and downloading will map to the same unit in a single-data-unit system. The question is how efficient it is when it incorrectly predicts the underlying systems, 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 one task and the other. This function is done automatically as sharing info is available only at the system level. Figure 7 shows the runtime analysis of an optimal solution for MIMO operation on a system with one DMA and one ALU using the hardware-ware and hardware-independent implementation. The result shows that although the hardware-independent model incorrectly predicts the underlying execution system, it still performs optimally.

4.5. 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 processing a single-image, no further change is required, and no synchronization is needed since each stream works on different data. However, it has several disadvantages:

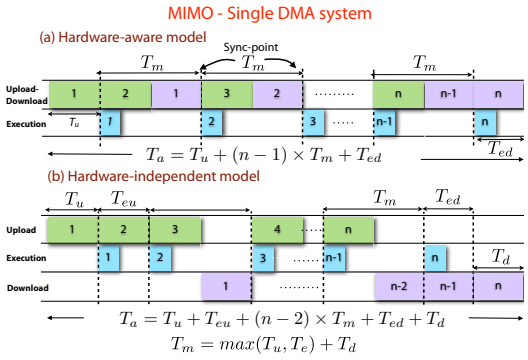

Figure 7: *Although the hardware-independent model mismatchs the system configuration, the performance is still optimal*

- The method does not reduce the memory usage and all the data must be loaded in-core. Hence, this method cannot be used for out-of-core processing.
- It requires the capability of decomposing input data and combining output results that is not always satisfied.
- 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. This estimation 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 the streams. Particularly, all streams execute in the same order of the logical flow: uploading-processing-downloading. However, flexible 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 much lower memory footprint, which is equal to the number of hardware devices with the hardware-aware model or number of decomposed tasks with the hardware-independent model. That means 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. That means the execution is controlled, providing several benefits. First, developers can profile the performance before they actually run it. Second, it reduces the complexity of scheduling problem to trivial mapping, so it is even opti-

mal without any automatic scheduler supports. And third, it helps to understand why degeneracy happens, how it affects the performance, and how to deal with it.

5. Re-ordering stages in streaming models

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

5.1. Forced Synchronizations

There are three primary reasons that degeneracies may 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 synchronization execution. Another reason is the mixed use of synchronous and asynchronous functions.

Even when all functions support asynchronous execution, they might be designed using different schemes. The strategies are often incompatible and cannot work together efficiently. For example, a kernel function defined to run on a logical stream, named 0, 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.

Cross-stream calls occur when the implementation requires data access and computation to or from different streams. As a result, the compiler force these streams to synchronize at cross-reference points to preserve the semantic order of the original program. One example is the traditional implementation of the class of reduction functions in CUDA. Though the computations run in-core on GPU-devices, the output of these functions, which are typically used for branching on CPU-host, require the result to be copied from device memory to host memory. This operation is a cross-stream function between the computational stream on the devices and the transfer data stream between devices and host. The popularity of the reduction functions is the main obstacle for applying asynchronous models on existing GPU architectures. Our solution for the reduction-like function is an on-device model that outputs the result only to device memory. It requires subsequent functions to use on-device parameters, and to delay or remove the branching in the codes.

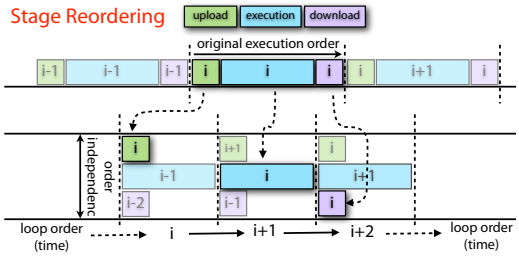

Figure 8: *The transformation from a synchronous model to an explicit streaming model preserves semantic correctness*

5.2. Re-ordering Pipeline Stages

Still, in many cases, a forced synchronization is unavoidable, though 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, to avoid data spilling, to keep the instruction pipelines filled, and especially to allow parallel execution on a system of multi-processors.

In this case of streaming with degeneracy, the reordering optimization cannot be done automatically using the compiler. The reason is that the uploading and downloading are the IO processes which have the 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. So reordering without compiler support must be done explicitly.

Allowing different streams working on independent images allows our explicit models to break the order-execution dependency inside the loop, replacing it with an equivalent order-independent streaming model. As shown in Figure 8, 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 becomes unimportant, we can change the order of streams at the API level from the regular order of upload-process-download to upload-download-process, or process-upload-download. The ability to change the processing order allows streaming optimization. This optimization is particularly effective when asynchronous stream degeneracy is unavoidable.

In the implicit model, when the synchronizations exists in the execution process, it is unable to overlap the uploading and downloading stream as the uploading process has to finish before the synchronization points, while the downloading only happens after the synchronization points. As shown on Figure 9, changing the order of streams in the code using the explicit model allows the upload and download stream

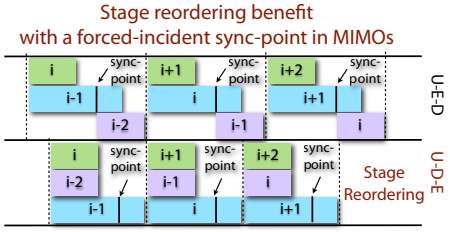

Figure 9: *Streaming optimization using reordering technique. As shown on the figure it is able to eliminate the negative effect of forced-synchronous function*

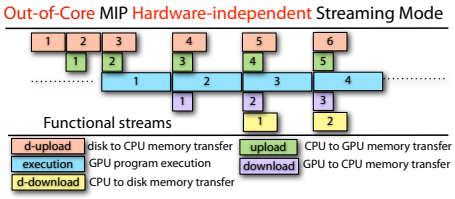

Figure 10: *The implementation of hardware-independent model for "full" out-of-core multi-image processing*

to be fully overlapped 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. Extension to a full out-of-core framework

The extension from the partial out-of-core model with one level of memory hierarchy to a full out-of-core model with a two-memory levels comes naturally with the hardware-independent model. By adding two more stages to the algorithm decomposition-the upload from disk to CPU memory and download from the CPU memory to disk-we realize the transition to a fully out-of-core model. The execution of this model for MIMO operation was displayed on Figure 10.

Using the same logic as the partial out-of-core model, we can prove that the hardware-independent model for out-of-core processing is optimal. Note that we use the term "full" to mean that the data could be stored on the hard drives of a single machine. However, our hardware-independent model could be further extended to the other out-of-core models, such as the data stream on the network and the system with higher memory hierarchy levels, and we can still prove that the proposed models are optimal.

7. Results

The system we used in our experiment 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.

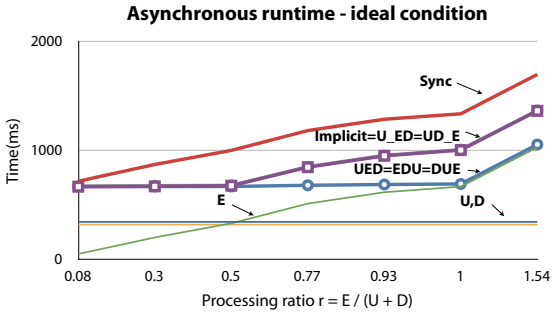

Figure 11: *Runtime comparison of different streaming strategies in ideal conditions.*

The program is compiled with CUDA NVCC 3.1. Run-time of each function is measured in milliseconds.

We made a synthetic test on a data set of 32 volumes, sized $256 \times 256 \times 256$. The test mimics a typical out-of-core multi-image 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 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).

The existing architecture on commodity hardware has single DMA unit, so the upload and download process has 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). 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, however, here we report the performance for three permutations: (1) regular upload-execution-download (UED) (2) execution-download-upload (EDU) (3) download-upload-execution (DUE).

7.1. Full asynchronous processing

First, we perform our test using the ideal cases, full asynchronous processing function, without a single synchronous call in the execution. Here we measure the influence of the ratio betwen computation and data transfer (processing ratio) on the performance of different asynchronous processing models, denote $r_e = E/(U+D)$. 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 11 show:

Function	U	E	D	Sync	Impl	Hrd-aware	Hrd-indp
Max	347	13	0	360	349	349	349
Energy	692	20	0	710	698	700	698
Averaging	347	20	11	378	360	363	361
Normalization	347	28	322	694	696	687	677
Gaussian	347	431	322	1099	735	770	678
Atlas	201446	213423	1359583	555204	NA	372567	340356

Table 1: *Runtime comparison of regular functions in practices with different streaming strategies.*

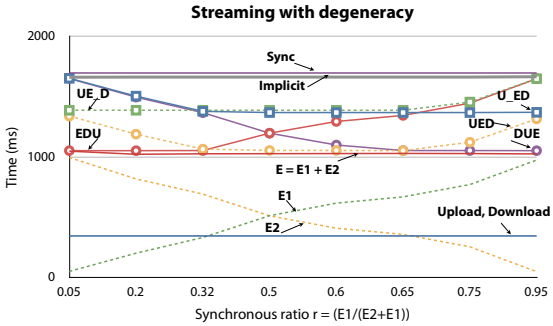

Figure 12: *Runtime comparison of different streaming strategies in degenerate conditions.*

- In all the tests, the three hardware independent implementations give us the same performance. The hardware-aware and implicit models give similar runtimes. The U_ED is slightly faster than UE_D since the upload takes a bit longer than the download.
- If the function is transfer-dominant ($r_e < 0.5$), all the models give optimal solutions.
- When the execution time is larger than the upload or the downloading time, the first two models still give strong performance, 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. The hardware-independent model is faster than the hardware aware model in this case 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 MIP out-of-core functions to outperform MIP 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 ($r_e = 1$).

7.2. Synchronous functions

Second, we test the result with the use of a synchronous function. Here we fix the run-time of three basic processes but change 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 synchonous ratio $r_s = E1/(E1+E2)$. With the existence of the synchronous function, the results in Figure 12 show:

- The position of the sync point within the asynchronous code directly affects the performance of the given implementations.
- The three hardware-independent implementations give us different performance characteristics. No single hardware-inpendent implementation gives us the best running time overall. However, the best result always is achieved with one of the hardware-independent 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 could not give us 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.

Though we show the results with execution-dominant function here, we also draw the same conclusions from transfer-dominant and balanced functions.

7.3. 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), 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. As the complexity of the functions increases (for example, Gaussian filtering function), we start seeing significant benefits of asynchronous streaming strategies, especially with the hardware independent model. 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 improve over the synchronous out-of-core version. The performance is compared to the in-core performance (execution time only) while we could process a significant amount of data much larger than that of an in-core version.

Overall, our 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 the tests. In the degenerate cases, the implicit model completely fails. An presence of synchronization points makes it impossible to find a efficient schedule 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 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.

8. Conclusions

In this paper, we have presented an optimized, parallel, multi-image processing framework on heterogeneous commodity systems extending from the existing single-image, parallel processing framework. We have introduced multi-image operators, serving as the connection between the single-image processing model and the multi-image processing variant. We proposed two basic multi-image operators: the MIMO and the MISO, which are utilized to construct other multi-image operators, allowing us to build a complete multi-image processing framework. We have also presented optimal streaming models for the multi-image processing framework. We have 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. Our 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 an quantitative evaluation that serves as a model for developers. We have investigated an optimal streaming strategy in unfavorable conditions based on reordering from order-independent properties of the explicit-streaming models. We also give insight 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 systems.

9. Appendix

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

Algorithm 1: Atlas construction framework

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

Algorithm 2: Synchronous out-of-core MIMO operators

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

Algorithm 3: Synchronous out-of-core MISO operators

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

Algorithm 4: Implicit pipelining MIMO operator

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

References

[Ala92] ALATTAR A.: A probabilistic filter for eliminating temporal noise in time-varying image sequences. In *Circuits and Sys-*

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

Algorithm 6: Explicit pipelining MISO operator

tems, 1992. ISCAS '92. Proceedings., 1992 IEEE International Symposium on (May 1992), vol. 3, pp. 1491 –1494 vol.3.

[BCK*95] BORDAWEKAR R., CHOUDHARY A., KENNEDY K., KOELBEL C., PALECZNY M.: A model and compilation strategy for out-of-core data parallel programs. *ACM SIGPLAN Notices 30*, 8 (1995), 1–10.

[BJW95] BOVE V. M., JR., WATLINGTON J. A.: Cheops: A reconfigurable data-flow system for video processing. In *IEEE Transactions on Circuits and Systems for Video Technology* (1995), pp. 140–149.

[BMK01] BROWN A., MOWRY T., KRIEGER O.: Compiler-based i/o prefetching for out-of-core applications. *ACM Transactions on Computer Systems (TOCS) 19*, 2 (2001), 170.

[Boy92] BOYCE J.: Noise reduction of image sequences using adaptive motion compensated frame averaging. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on* (Mar. 1992), vol. 3, pp. 461 –464 vol.3.

[BWP04] BITTNER J., WIMMER M., PIRINGER... H.: " coherent hierarchical culling: Hardware occlusion queries made useful"; computer graphics forum, 23 (2004), 3; s. 615-624. *Computer Graphics ...* (Jan 2004).

[CDS05] CARON E., DESPREZ F., SUTER F.: Out-of-core and pipeline techniques for wavefront algorithms. *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers 01* (Apr 2005).

[CESL*03] CHIANG Y., EL-SANA J., LINDSTROM P., PAJAROLA R., SILVA C.: Out-of-core algorithms for scientific visualization and computer graphics. *IEEE Visualization* (2003).

[CMVG96] CHRISTENSEN G. E., MILLER M. I., VANNIER M. W., GRENANDER U.: Individualizing neuroanatomical atlases using a massively parallel computer. In *Computer* (1996), vol. 29, IEEE Computer Society, pp. 32–38.

[DFBJ07] DAVIS B., FLETCHER P., BULLITT E., JOSHI S.: Population shape regression from random design data. *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on* (Oct 2007), 1–7.

[DM95] DUFAUX F., MOSCHENI F.: Motion estimation techniques for digital tv: a review and a new contribution. *Proceedings of the IEEE 83*, 6 (June 1995), 858 –876.

[EAK10] EKLUND A., ANDERSSON M., KNUTSSON H.: Phase based volume registration using cuda. In *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on* (Mar. 2010), pp. 658 –661.

[Fly72] FLYNN M. J.: Some computer organizations and their effectiveness. *Computers, IEEE Transactions on C-21*, 9 (Sep 1972), 948 –960.

[GSC*07] GOESELE M., SNAVELY N., CURLESS B., HOPPE H., SEITZ S.: Multi-view stereo for community photo collections. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on* (Oct 2007), pp. 1 –8.

[HE07] HAYS J., EFROS A. A.: Scene completion using millions of photographs. *ACM Trans. Graph. 26* (July 2007).

[HKF*09] HA L. K., KRÜGER J., FLETCHER P. T., JOSHI S., SILVA C. T.: Fast parallel unbiased diffeomorphic atlas construction on multi-graphics processing units. In *EUROGRAPHICS Symposium on Parallel Graphics and Visualization 2009* (2009).

[HKJS11] HA L., KRÜGER J., JOSHI S., SILVA C. T.: *Multiscale Unbiased Diffeomorphic Atlas Construction on Multi-GPUs*, vol. I. Elsevier, Jan 2011.

[HYWL07] HU C., YAO G., WANG J., LI J.: Transforming the adaptive irregular out-of-core applications for hiding communication and disk i/o. *OTM'07: Proceedings of the 2007 OTM confederated international conference on On the move to meaningful internet systems: CoopIS, DOA, ODBASE, GADA, and IS Part II* (Nov 2007).

[JDJG04] JOSHI S., DAVIS B., JOMIER M., GERIG G.: Unbiased diffeomorphic atlas construction for computational anatomy. *Neuroimage 23 Suppl. 1* (2004), S151–S160.

[Lam79] LAMPORT L.: How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput. 28* (September 1979), 690–691.

[Mac94] MACOVKSI A.: Tolerating latency through software-controlled data prefetching. *en.scientificcommons.org* (Jan 1994).

[MDK96] MOWRY T., DEMKE A., KRIEGER O.: Automatic compiler-inserted i/o prefetching for out-of-core applications. *ACM SIGOPS Operating Systems Review 30*, si (1996), 3–17.

[Meu10] MEUER H.: China grabs supercomputing leadership spot in latest ranking of world's top 500 supercomputers. http://www.top500.org, Nov. 2010.

[RDK*98] RIXNER S., DALLY W. J., KAPASI U. J., KHAILANY B., LÓPEZ-LAGUNAS A., MATTSON P. R., OWENS J. D.: A bandwidth-efficient architecture for media processing. In *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture* (Los Alamitos, CA, USA, 1998), MICRO 31, IEEE Computer Society Press, pp. 3–13.

[RSM10] ROBERTS M., SOUSA M. C., MITCHELL J. R.: A work-efficient gpu algorithm for level set segmentation. In *ACM SIGGRAPH 2010 Posters* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 53:1–53:1.

[SGSS08] SNAVELY N., GARG R., SEITZ S. M., SZELISKI R.: Finding paths through the world's photos. *ACM Trans. Graph. 27* (August 2008), 15:1–15:11.

[SRC09] SUNDARAM N., RAGHUNATHAN A., CHAKRADHAR S. T.: A framework for efficient and scalable execution of domain-specific templates on gpus. *Parallel and Distributed Processing Symposium, International 0* (2009), 1–12.

[SSS06] SNAVELY N., SEITZ S. M., SZELISKI R.: Photo tourism: Exploring photo collections in 3d. In *SIGGRAPH Conference Proceedings* (New York, NY, USA, 2006), ACM Press, pp. 835–846.

[WGRW02] WOMBLE D., GREENBERG D., RIESEN R., WHEAT S.: Out of core, out of mind: Practical parallel i/o. *Scalable Parallel Libraries Conference, 1993., Proceedings of the* (2002), 10–16.