

# Scalable Framework for Mapping Streaming Applications onto Multi-GPU Systems

Huynh Phung Huynh

A\*STAR Institute of  
High Performance Computing  
Singapore

huynhph@ihpc.a-star.edu.sg

Andrei Hagiescu<sup>1</sup> Weng-Fai Wong

School of Computing  
National University of Singapore  
Singapore

{hagiescu,wongwf}@comp.nus.edu.sg

Rick Siow Mong Goh

A\*STAR Institute of  
High Performance Computing  
Singapore

gohsm@ihpc.a-star.edu.sg

## Abstract

Graphics processing units leverage on a large array of parallel processing cores to boost the performance of a specific streaming computation pattern frequently found in graphics applications. Unfortunately, while many other general purpose applications do exhibit the required streaming behavior, they also possess unfavorable data layout and poor computation-to-communication ratios that penalize any straight-forward execution on the GPU. In this paper we describe an efficient and scalable code generation framework that can map general purpose streaming applications onto a multi-GPU system. This framework spans the entire core and memory hierarchy exposed by the multi-GPU system. Several key features in our framework ensure the scalability required by complex streaming applications. First, we propose an efficient stream graph partitioning algorithm that partitions the complex application to achieve the best performance under a given shared memory constraint. Next, the resulting partitions are mapped to multiple GPUs using an efficient architecture-driven strategy. The mapping balances the workload while considering the communication overhead. Finally, a highly effective pipeline execution is employed for the execution of the partitions on the multi-GPU system. The framework has been implemented as a back-end of the StreamIt programming language compiler. Our comprehensive experiments show its scalability and significant performance speedup compared with a previous state-of-the-art solution.

**Categories and Subject Descriptors** D.3.4 [Programming languages]: Processors—Code generation

**General Terms** Algorithms, Performance, Design

## 1. Introduction

The interest in using Graphics Processing Units (GPUs) for general purpose computation has expanded beyond the high performance computing community [19, 20] into mainstream computing. GPUs are parallel processors that consist of a number of *streaming multi-processors* (SM), which in turn are made up of a number of process-

ing cores running in lockstep. They support the execution of computational *kernels* – blocks of threads executed together on each of the available SMs. Massive parallelism is achieved as a large number of threads is interleaved onto the smaller number of processing cores. In order to ease the task of GPU developers, several GPU programming models have been proposed, such as OpenCL [16] and CUDA [1].

General purpose streaming applications are suitable for GPU processing as they expose significant task and data-level parallelism. Hence, several stream programming languages were proposed to ease the expression of parallelism in such applications [4, 8]. These languages capture computation in the form of a stream graph, where the graph nodes can consist of self-contained tasks, or finer-grained compute elements. StreamIt [8] is a platform-independent stream programming language that distinguishes among the alternative options. It is suitable for compilation onto a large number of platforms due to the fine-grained parallelism it exposes. It has also proved to be a suitable candidate for GPU compilation [11, 12, 23].

Usually, stream graphs are compiled and the code from the resulting nodes is mapped individually as GPU *kernels*. The code in each kernel expresses how a number of consecutive node executions are processed by the architecture. Multiple executions are handled in parallel, in distinct GPU threads. The adjacent nodes in the stream graph communicate through *channels*, currently implemented as FIFO queues in memory. At the beginning of a node execution, input data is read from a channel, and at the end of the execution, the output data is written to a different channel, all in a phased manner. Conventional GPU mapping may place these channels in either the GPU global memory, or the faster but size-limited SM memory<sup>2</sup>.

One straightforward implementation strategy is to place channels in the global memory. As GPU code tend to instantiate a large number of threads, the hardware scheduler is relied upon to interleave these threads so as to hide potential stalls caused by global memory accesses [1]. Unfortunately, increasing the number of threads will decrease the number of registers allocated to each of them, potentially causing additional spills to global memory. In addition, more threads will require more input, output and local data stored in the global memory, increasing further memory traffic, and saturating the available memory bandwidth rapidly. For streaming applications, in particular, the intensive memory access at the beginning and end of each node execution that accesses the channels can exacerbate these problems.

<sup>1</sup>The first and second authors contributed equally to this paper

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.  
Copyright © 2012 ACM 978-1-4503-1160-1/12/02...\$10.00

<sup>2</sup>The nVidia way of referring to this as *shared memory* is potentially confusing.

On the other hand, if the channels are placed in the fast SM memory, the number of parallel executions is limited by the small size of this memory. The resulting thread pool may underutilize the GPU processing cores. If the computation-communication ratio is low, this also undermines the utilization of any prefetching techniques.

To overcome the issues mentioned above, an automated GPU mapping framework for StreamIt applications [11] has been proposed. That framework instantiated a mixed pool of compute and memory access threads that maximized the utilization of the SM memory available. The resulting computation structure also maximized the utilization of the GPU cores given the memory constraint. However, this implementation is limited to a single partition. Hence, the application performance was severely limited if the working set size grew large enough such that insufficient parallel threads could be instantiated in the SM memory.

In this paper we present a mapping strategy that can handle complex streaming applications where the overall memory requirement for efficient implementations is larger than the available SM memory. This is achieved through an efficient graph partitioning algorithm that splits the application into several smaller partitions that satisfy the SM memory constraint, hence achieve good performance individually.

Scalability of our method is demonstrated by the ability to distribute the resulting partitions on a multi-GPU system. In addition, one or more partitions can be mapped to each GPU, resulting in a combined spatial and temporal distribution. The strategy also identifies graph nodes that are not suitable for GPU implementation, and map these nodes to CPU cores. Furthermore, the communication overhead between partitions is considered during the mapping step. This is necessary because a high volume of data streamed between partitions has to pass through one or more of the slower levels of the memory hierarchy, as determined by the locality of the partitions.

The following are the key contributions of this work:

- a scalable code generation framework that handles complex StreamIt applications (Section 4)
- an algorithm that divides large applications onto several partitions valid under SM memory constraints (Section 5).
- a multi-GPU mapping and orchestration scheme for these partitions (Section 6).

To the best of our knowledge, this is the first work on mapping StreamIt onto a single compute node that has multiple GPUs. The comprehensive set of experiments shows that, despite the fact that the application is not trivially parallel, our method was able to significantly improve performance by utilizing multiple GPUs.

## 2. Related Work

The StreamIt programming language [8] is based on synchronous data flow graphs [18]. It has been compiled on a myriad of architectures including multi-cores [9], Cell [17], Raw [8] and even FP-GAs [10]. Moreover, there is previous research in coarse-grained mapping of StreamIt to GPU platforms [23].

The GPU mapping usually involves kernels that communicate to each other through global memory. Therefore, the overall performance is limited by the high latency of accessing this memory [24]. Recently proposed methods [11, 12] improved the memory access scheme by using two classes of dedicated threads for: (1) loading / storing data from global memory to SM memory and (2) computing using data preloaded in SM memory. The two methods contrast in how computation is organized. One exploits the coarse-grained task parallelism exposed by StreamIt graph nodes [12], leaving behind significant data-parallel optimizations opportunities available

in each stream graph schedule execution. The other took advantage of finer-grained data level parallelism opportunities, where a single execution of the stream graph schedule spans several computing threads [11]. However, the latter method does not scale well with large applications because it attempts to map the entire stream graph schedule to a single partition.

Besides utilizing multiple partitions, a promising solution to deal with the scalability issue is the utilization of multi-GPU systems. Such systems are well-suited to process large data set applications [22]. Execution on multi-GPU systems has been improved through various run-time schemes, such as load balancing [5] and speculative execution [6]. Many of these schemes can be complemented by static methods that estimate the performance of an application running on the multi-GPU system, based on characterizing computation and different communication costs [21]. Performance modeling for GPU architectures was comprehensively investigated by analytic and quantitative approaches [3, 25] which highlighted the important balance between computation and memory access, as well as the utilization of SM memory. However, none of those works has attempted to map applications automatically, nor to provide an execution model for streaming languages onto multiple GPUs in an integrated approach.

Given the above challenges, this multi-GPU mapping work will also tackle the scalability of the solution [11], by partitioning the StreamIt application before mapping it. This is the first mapping attempt that integrates optimal GPU kernel generation, load-balanced mapping, communication reduction, and pipelined orchestration for multi-GPUs systems.

## 3. Background

### 3.1 StreamIt programming language

StreamIt was designed to expose the parallel and pipelined nature of the streaming applications. The high-level structure of a StreamIt program is a graph whose basic nodes are *filters* which communicate through channels. Filters can be combined to execute in *pipelines*. The data flow of the filters can also be distributed using *splitters* and *joiners* that describe parallel execution paths in the application. These two constructs expose the coarse-grained task parallelism in the application.

Filters are written in C-like code with special constructs to access their input and output channels. A filter consumes data from an input channel using *pop* constructs, and produces data on the output channel using *push* constructs. The input (output) rates between two filters may be different but they are statically defined. Therefore, multiple filter executions may be required to match the rates between filters. This exposes fine-grained data parallelism in the streaming application. The StreamIt compiler takes into account the non-matching rate of input (output), as well as dependencies among filters in the stream graph, to generate a static schedule for the entire graph. The static schedule contains *operators* (filters, splitters, joiners), which can be iteratively executed to process all the incoming data. Note that multiple copies of the schedule can be executed in parallel to process different data segments, as long as the filters in the schedule do not maintain internal state. The filters also have the capability to *peek* data beyond what they are going to consume through *peek* constructs. This feature allows structured data dependencies between consecutive filter executions and do not restrict parallel execution.

### 3.2 Mapping onto GPUs

The number of processing cores in streaming multiprocessors (SM) continues to increase with each new generation of GPUs (up to 48 cores per SM in the most recent nVidia GPUs, compared to S2050's 32 cores and S1070's 16 cores). Therefore, the number of threads

supported in each SM also increases proportionally. The threads of each SM are divided into groups of 32 threads, called *warps*. Warp executions are interleaved by a hardware scheduler on the small number of processing cores. All threads in the same warp execute the same instruction at each time step (lockstep execution, a variant of SIMD) and divergent flow is serialized. However, threads that belong to different warps are independent of any divergent flow penalty. The hardware scheduler selects a warp for execution and dispatches the instruction from the selected warp to the processing cores. While the instruction is propagated through the execution pipeline, or while it waits for memory operands, the scheduler switches to execute a different warp with zero-overhead. As a result, though a large number of parallel threads can be spawned, their executions are actually interleaved on the processing cores.

Another reason behind instantiating a large number of threads is to hide the long delay of GPU global memory access through the interleaving mechanism described above. When mapping StreamIt applications onto GPUs, global memory access becomes more frequent because fine-grained operators have to communicate to each other through memory channels. Even if SM memory is used to prefetch the data from global memory, the bottleneck is still visible because operators seldom reuse the data they read from the channels. Previous work [11] describes a scheme that reduces the pressure on global memory access for streaming applications. First, several operators can be executed in the same GPU kernel, where they communicate through SM memory. Then, two distinct classes of threads are instantiated: memory access ( $\mathcal{M}$ ) threads and compute ( $\mathcal{C}$ ) threads.  $\mathcal{M}$  threads are only responsible to prefetch data from global memory to SM memory while  $\mathcal{C}$  threads only perform computations on the data prefetched by  $\mathcal{M}$  threads. An efficient heuristic is used to determine the number of  $\mathcal{C}$  and  $\mathcal{M}$  threads as well as the execution schedule for those threads. The data-level parallelism available inside the stream graph is analyzed, such that entire executions of the stream graph are mapped onto multiple  $\mathcal{C}$  threads inside an SM. This scheme is replicated on all the other SMs to fully utilize the GPU.

Such a mapping flow is feasible if the memory requirement for the parallel stream graph executions is less than or equal to the capacity of SM memory. Increasing memory requirement from large data-set and complex streaming applications, would result in a reduction of the number of supported parallel executions. The alternative investigated by this paper is to split the stream graphs of those applications into smaller partitions whose memory requirements match the SM memory constraint. To enable further scalability, we investigate the mapping of those partitions onto a multi-GPU platform. Altogether, we propose a scalable mapping framework for mapping streaming applications onto GPUs. The overview of the framework is presented in the next section.

#### 4. Scalable Mapping Framework

The scalable mapping framework proposed in this paper is based on a recent work on automated GPU mapping for StreamIt applications [11]. The input to the framework are the applications written in the StreamIt language. The StreamIt compiler [8] analyzes the input program to generate a schedule of operators, as well as perform some common optimizations. It is the stream graph expressed in the schedule that is the input for our framework.

The components that form this framework, shown in shadows in Figure 4, provide the following:

- A stream graph partitioning algorithm that breaks complex StreamIt applications into smaller partitions that utilize the SMs efficiently.
- A global mapping step that balances the partitions among the available GPUs.

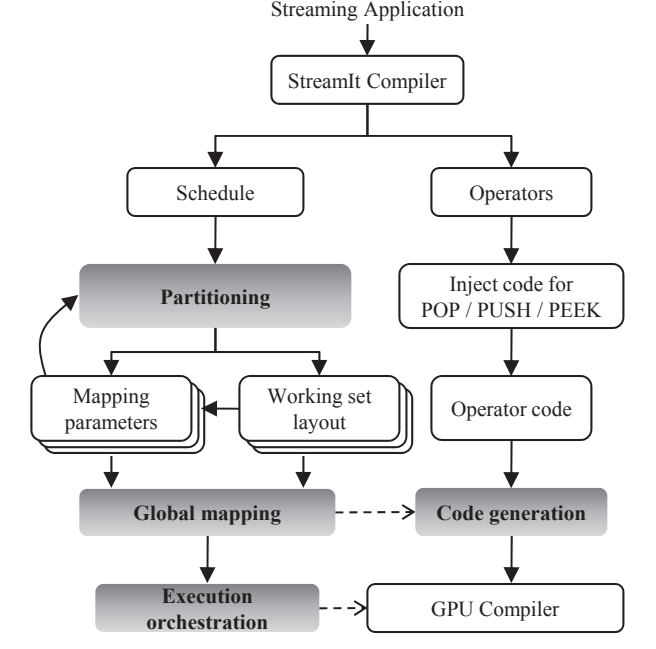


Figure 1. Scalable Mapping Framework.

- A code generator that provides C code for the partitions, as well as code coordinating the communication between them, based on the mapping result.
- A pipelined execution orchestration for the partitions.

The stream graph partitioning component prunes the design space by analysing the validity and estimated performance of the possible partitions (details in Section 5). The performance estimation considers the specification of the target GPU. The result of this partitioning is a set of convex and disjoint sub-graphs which are ready for mapping to GPU. Operators that maintain internal state are included in separate partitions that are executed on the CPU cores. For each GPU partition, we compute a compact memory layout [11] that can be realized in the fast SM memory. Given the memory layout for each partition, the other parallel code mapping parameters are determined by heuristics [11]. These heuristics depend on the specifications of the target GPU.

The push, pop and peek primitives of each operator in the schedule are annotated with information about how the channels in the SM memory are to be accessed correctly. These annotation will be used during the automatic code generation process.

Finally, the resulting set of partitions is passed to a global mapping step which assigns each partition to a specific GPU or CPU core. At this stage, communication channels between partitions are also instantiated (details in Section 6). The generated code is orchestrated by an execution environment which contains: (1) a multi-threaded controller which will run on the host CPU, and (2) the inter-partition memory communication scheme for pipelined execution. The controller consists of threads that coordinate the kernels loaded on each GPU, as well as threads that execute the CPU partitions.

#### 5. Stream Graph Partitioning

Given a stream graph, the objective of partitioning is to maximize the overall performance of the stream graph, while ensuring that the partitions satisfy resource constraints, and yet effectively utilize the GPU.

## 5.1 Definitions

A stream graph  $G(V, C)$  represents the data flow within the stream application. The nodes  $V$  represent the *operators*, and the edges  $C$  represent the channels (dependencies) between the operators. A *channel* connects the output of a producer operator to the input of a consumer operator. As advanced features of StreamIt such as feedback loops and portals are not supported,  $G(V, C)$  is always a directed acyclic graph.

A partition  $P$  must be a convex subgraph, as non-convex subgraphs cause heavy communication to the adjacent subgraphs. Even worse, they may lead to deadlocks.  $P$  is convex if there does not exist a path in  $G(V, C)$  from an operator  $V_m \in P$  to another operator  $V_n \in P$ , which contains an operator  $V_p \notin P$ .

## 5.2 Partitioning Algorithm

The method employed to identify suitable partitions relies on the well-known  $k$ -way graph partitioning algorithm [14]. In this algorithm, the nodes of a graph are partitioned into  $k$  roughly equal partitions so that the weight of the edges between nodes in different partitions (edge-cut) is minimized. Intuitively, this results in load balanced partitions that have minimum communication. However, our partitioning differs from the standard algorithm in several aspects: (1) the number of partitions  $k$  is not an input to the problem – the value of  $k$  will be determined during the run-time of the algorithm such that it maximizes the overall performance, (2) there are convexity and memory constraints on each partition – these constraints affect the performance of combined partitions, and (3) the objective of the algorithm is to maximize the performance of the application. The performance objective is estimated as  $\sum_{i=1 \dots k} T(P_i)$

where  $T$  is an estimation of the execution time of  $P_i$ . Even if multiple GPUs are utilized, a balanced distribution of the partitions to the multiple GPUs ensures that this objective continues to reflect the overall stream graph execution performance.

Nevertheless, the *multi-level graph partitioning* (MLGP) algorithm used to solve the  $k$ -way problem can be effectively employed to solve our problem. In MLGP, the nodes in the original graph are grouped to create coarser nodes (the *coarsening phase*). The original graph is iteratively coarsened down to  $k$  partitions, over a number of levels, in order to create the initial partitioning solution (the *partitioning phase*). Then, the initial solution is *uncoarsened* back to the original graph by using the same number of levels as in the coarsening phase. While uncoarsening, the partitioning solution is refined by the movement of nodes to adjacent partitions so as to improve the overall performance.

We adapt an efficient multi-level algorithm [14] to our graph partitioning problem. This approach has also been effectively used in other contexts [13]. In our approach we continue to decrease the number of partitions as long as overall performance of the entire stream graph is still increasing. Because we do not have a particular  $k$  value as the input to our graph partitioning, we eliminate the  $k$ -partitioning limit from the MLGP algorithm. Alternatively, the number of partitions of the solution is the number of nodes in the coarsest graph obtained. The details of the coarsening and uncoarsening phases are described below.

## 5.3 Coarsening phase

From the original directed stream graph, we create a sequence of coarser graphs  $G_i = (V_i, C_i)$  by clustering together pairs of nodes. A node  $u \in V_{i+1}$  in a coarsened graph  $G_{i+1}$  at level  $i + 1$  is the result of merging two *matching* nodes  $v, w \in V_i$  of the finer graph  $G_i$  at level  $i$  such that  $u$  is convex and can be implemented on the GPU. Otherwise, if we can not find a convex combination, node  $u$  is simply set to vertex  $v \in V_i$  of  $G_i$ . Note that each node  $u$  in a coarse graph is a sub-graph of  $G_0$  when projected from the

constituent nodes of  $u$  in the finer graph. In  $G_2$  of Figure 2, the sub-graph corresponding to coarse vertices  $\{0,1\}$  consists of vertices  $\{0,3,5,6,7,9\}$  of  $G_0$ . After constructing coarser nodes, we build the edges  $C_{i+1}$  of the coarse graph. There is a directed edge between two nodes in coarser graph  $G_{i+1}$  if there exists a directed edge between the constituent nodes in the finer graph  $G_i$ .

In our matching heuristics, the nodes of  $G_i$  are visited in random order. We select an unmatched node  $v \in V_i$ , and then iterate through the unmatched nodes adjacent to  $v$  to find a possible match  $w$  under the convexity constraint. We consider only adjacent nodes because there will be significant communication overhead among coarse nodes if we merge non-adjacent nodes. Two nodes are merged if their matching yields the best performance gain. The performance gain here is defined as:  $\Delta T = T(w) + T(v) - T(u)$ .

The estimation of  $T$  is based on the amount of SM memory required by the subgraph executions, because this determines the number of subgraph executions that can be run in parallel. The SM memory requirement is derived through channel layout analysis. While previous work employs an algorithm which considers the influence of fragmentation on the channel layout [11], the complexity of the partitioning algorithm can be reduced by using an estimation which does not consider the effect of fragmentation.

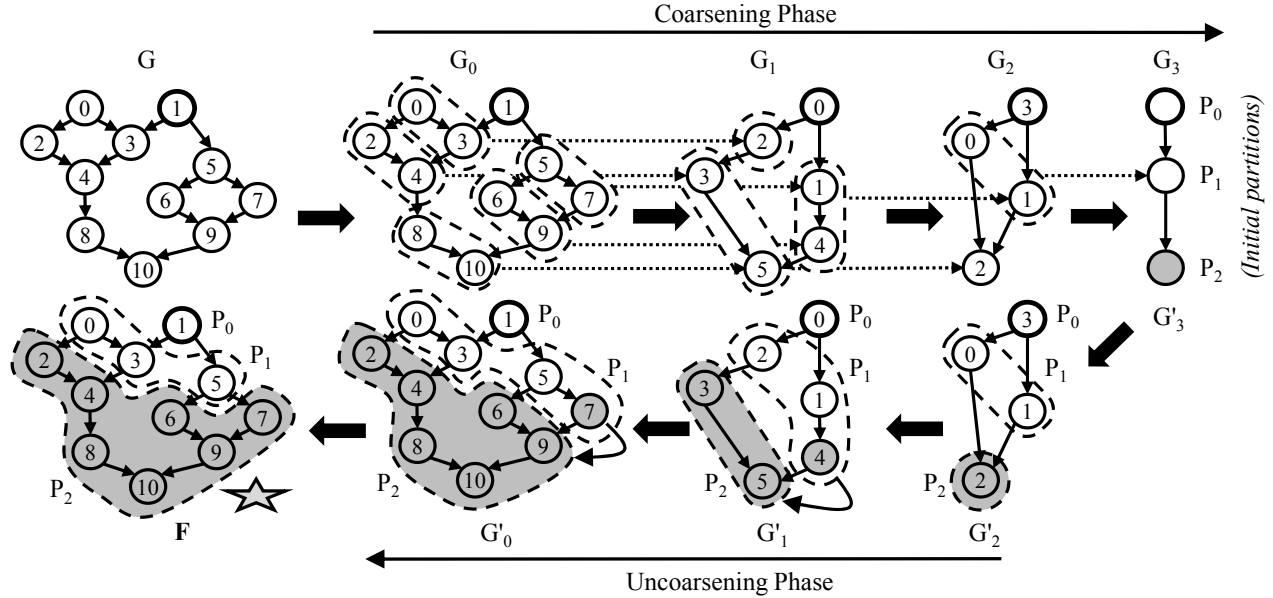
In case a feasible matching for  $v$  can not be found,  $u$  inherits only a single node  $v$ . In Figure 2, nodes 1 and 4 of  $G_1$  are matched to form node 1 of  $G_2$  while node 2 of  $G_1$  is assigned to node 0 of  $G_2$ . Note that the filters that maintain state are more suited for CPU execution (i.e. node 1 in  $G_0$ ), because they can not be parallelized. Therefore, we prevent these filters from being matched as they will be included in special partitions to be mapped to CPU cores.

If the graph cannot be coarsened any further, i.e.,  $G_{i+1} = G_i$ , the coarsening phase ends. Let  $G_m = (V_m, C_m)$  be the coarsest graph achieved. The initial partitioning solution utilizes this configuration, and each node  $v \in V_m$  is selected as a partition. The number of partitions,  $k$ , is just  $|V_m|$ . This value is not an input as is the case in the standard  $k$ -way problem, but it is only determined when the coarsest graph is reached. These initial partitions will be refined as we go through the uncoarsening phase back to  $G_0$ . In Figure 2, the coarsening phase goes through a sequence of coarse graphs  $\{G_0, G_1, G_2, G_3\}$  and the initial coarsening leads to three partitions  $P_0, P_1$  and  $P_2$ .

## 5.4 Uncoarsening Phase

From the coarsest graph  $G_m$ , the initial partitions are projected back to the original graph by traversing a sequence of finer graphs  $G'_{m-1}, \dots, G'_0$ , where  $G'_i$  is a refinement of  $G_i$ . During this uncoarsening process, we need to trace the partition to which the finer nodes belong. Let  $P(v)$  be the partition assignment for a node  $v$ . Each node of the coarsest graph  $G_m$  represents a partition, so, utilizing this notation,  $P(v_i) = P_i$  ( $v_i \in V_m$ ). Because nodes in a level  $i + 1$  graph include one or two nodes from the level  $i$  graph, the partitioning information can easily be propagated through all the levels.

Moving nodes from one partition to another may yield improvements.  $G_j$  is less coarse than  $G_{j+1}$ . Therefore, there is more freedom to move the nodes in  $G_j$ . The movement may reduce the communication or increase the combined performance of the partitions after the move. There are local movement heuristics based on the Kernighan-Lin (KL) [15] or Fiduccia-Mattheyses (FM) [7] partitioning algorithms which can yield good results for bi-partitioned graphs. However, using the KL or FM methods in a  $k$ -way problem leads to significant complexity, because a node from a partition can move to several other partitions. Instead, for our problem, we developed a simple and efficient movement algorithm inspired from a greedy refinement method [14].



**Figure 2.** Illustration of Multi-Level Graph Partitioning. The dashed lines show the projection of a vertex from a coarser graph to a finer graph.

Our movement algorithm tries to move the boundary nodes of a partition to the adjacent partitions. A boundary node of partition  $P_i$  in coarse graph  $G_j = (V_j, E_j)$  is a node  $v \in V_j$  that has at least one adjacent node  $u \in V_j$  that belongs to a different partition ( $P(v) \neq P(u)$ ). In Figure 2, nodes  $\{2,4,6,9\}$  in  $G'_0$  are the boundary nodes of partition  $P_2$ , while  $\{8,10\}$  are the internal ones. We randomly select a boundary node  $v$  to move from partition  $P(v)$  (the *source partition*) to the neighborhood partitions  $P(u)$  (the *destination partition*). For  $G'_0$  in Figure 2, a neighborhood partition of node 7 is  $P_2$ . After moving node  $v$  from source partition  $P(v)$  to the destination partition  $P(u)$ , if the source and the destination partitions still satisfy the convexity and SM memory constraints, the movement is deemed valid and would transform the two original partitions  $P(v)$  and  $P(u)$  into the new partitions  $P(v)'$  and  $P(u)'$ . Among the valid movements of the boundary node  $v$  to the neighborhood partitions, the movement which has the highest  $\Delta T = T(P(v)) + T(P(u)) - T(P(v)') - T(P(u)')$  is selected and node  $v$  is moved to that particular destination partition. The source and destination partitions are updated respectively. The moved nodes will not be considered again for analysis during the current coarsening level. The movement algorithm for the current level stops if there are no more boundary nodes to move. Once the movement algorithm for  $G_i$  finishes, the uncoarsening phase continues by projecting back to  $G_{i-1}$ . Finally, after  $G'_0$  is analysed, we obtain the final assignment of the nodes to partitions. In Figure 2, node 4 is moved from  $P_1$  to  $P_2$  when  $G'_1$  is analysed, and node 7 is moved when  $G'_0$  is analysed. The result is the partitioned graph  $F$ , which captures the refinements to the partitioning solution.

## 6. Execution on a multi-GPU platform

After the original stream graph is partitioned, we need to map the partitions onto the multiple GPUs and CPU cores such that the workload is balanced. In this section, we first describe how such a balanced mapping is achieved. We will also describe the execution model that ensures the efficient utilization of the mapped partitions.

### 6.1 Partition Mapping

Each partition that belongs to the solution  $F$  obtained in Section 5 forms a single node in the coarsest graph  $G_m$ . An edge between any two level  $i+1$  nodes exists if there is an edge connecting constituent nodes of those two coarse nodes in  $G_i$ . The edge is assigned a *weight* which is the sum of the communication overhead of the level  $i$  edges. The communication overhead is the ratio between the data amount exchanged by two level  $i+1$  nodes, and the memory transfer bandwidth between CPU and GPU. In order to map  $G_m$  onto a system with  $x$  GPUs, we need to distribute the partitions in  $G_m$  onto  $x$  processing elements with the objectives: (1) the load should be balanced, and (2) the overhead of the communication edges should be minimized. The  $k$ -way partitioning algorithm is a good match for this problem because it splits the nodes of a graph into  $x$  roughly equal partitions such that the communication between the different partitions is minimized.

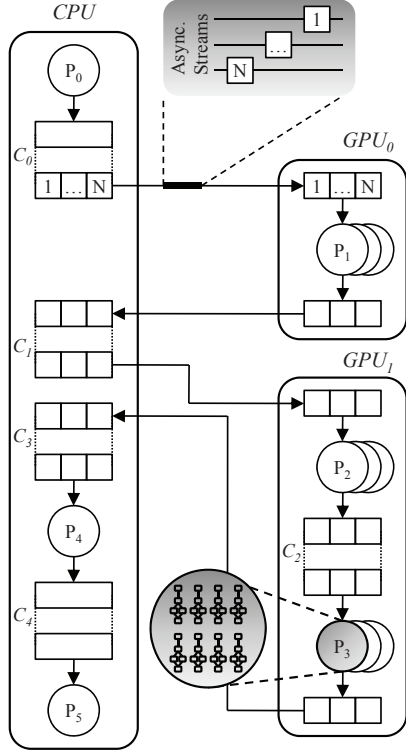
An exception are the partitions that maintain internal states. They correspond to individual filters due to the coarsening restrictions from Section 5. These partitions are pre-mapped to CPU threads and are not included in this second  $k$ -way partitioning pass. The remaining partitions are analysed, and divided among the  $x$  GPUs.

### 6.2 Communication channels

After mapping, each GPU has to multiplex the execution of the several partitions assigned to it. The code for a GPU kernel corresponding to each partition is generated as described in Section 3.2, with additional code inserted to assist with the pipelined execution of the partitions. The execution schedule on each GPU is coordinated by a dedicated CPU thread. Additional CPU threads are launched to support the CPU partitions.

The entire execution schedule utilizes memory-based FIFO (First In First Out) channels for data transfer. The FIFO length ensures that there will be no stall during the execution. Each FIFO element contains data corresponding to a large number of stream executions. This coarse data granularity takes advantage of the exposed data parallelism, and hides the unnecessary overhead of

handling data separately for independent iterations within a partition. Overall, three levels of data transfer are employed between: (1) the different partitions, (2) the asynchronous launches of the partition kernel using GPU streams, and (3) the compute  $\mathcal{C}$  and memory access  $\mathcal{M}$  threads inside each GPU partition.



**Figure 3.** Execution and data transfer among partitions on multiple GPUs

**Level 1 data transfer** We spawn  $x + y$  CPU threads to manage the parallel execution on the  $x$  GPUs and the  $y$  additional threads supporting CPU partitions. CPU synchronization primitives ensure uncorrupted access to the channels between the partitions. The FIFO channels between two CPU partitions or two partitions executed on the same GPU employ a standard circular buffer where memory pointers are passed directly between the threads.

However, when data needs to be transferred between CPU and GPU partitions, an additional buffering scheme that copies the channel data from CPU/GPU memory to GPU/CPU memory is used. For example, in Figure 3, the data in channel  $C_3$  between partition  $P_3$  (on  $GPU_1$ ) and partition  $P_4$  (on CPU) requires this buffering scheme.

Finally, in order to transfer data between partitions on different GPUs, the data is always copied first to the CPU, where the FIFO channel is implemented. Afterwards, data is copied from CPU memory to the other GPU using the buffering scheme described above. In Figure 3, the output buffer of partition  $P_1$  (in  $GPU_0$ ) is copied to the channel  $C_1$  in CPU memory and data from this channel is copied to the input buffer of partition  $P_2$  (in  $GPU_1$ ).

Our communication scheme between partitions on different GPUs can be easily adapted to the recent peer-to-peer memory access in CUDA 4.0 [1]. Note that peer-to-peer memory access is specific to nVidia GPUs. Moreover, in order to use peer-to-peer memory access in our pipeline execution, we still need to perform synchronization among different CPU threads to ensure that memory accesses are uncorrupted. More importantly, peer-to-peer

memory copy between two GPUs can not be initiated until all commands previously issued to either GPU have completed, and has to complete before any asynchronous commands issued after the copy to either GPU can start. This may downgrade the benefit of peer-to-peer communication in comparison with communication through CPU, which can benefit from the support of asynchronous GPU streams.

**Level 2 data transfer** The asynchronous streaming support for the GPUs is utilized to hide the CPU/GPU memory copy overhead. The coarse data elements from the FIFO channels are divided into smaller fragments. We generate an asynchronous stream of memory copy and partition kernel launch requests to process the GPU copy and execution. As the operations on these fragments are independent, memory transfers and kernel executions of different fragments can overlap.  $N$  fragments are created, as shown in Figure 3. Each stream will coordinate data transfer and execution for its corresponding fragment. While Stream 1 is performing computation of fragment 1 in  $GPU_0$ , Stream 2 can transfer fragment 2 from CPU to  $GPU_0$ .

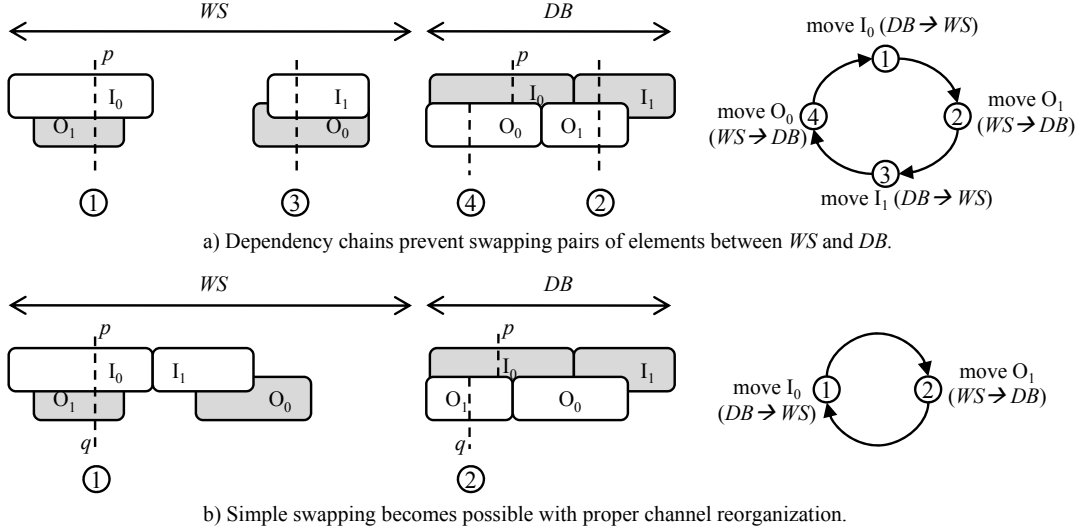
If several partitions are mapped to a single GPU, these partitions are time multiplexed. In Figure 3, the execution of partition  $P_3$  and  $P_2$  are interleaved.

**Level 3 data transfer** Each GPU kernel executes multiple iterations over the group of parallel executions of the stream graph partition it includes [11]. Using a mix of  $\mathcal{C}$  and  $\mathcal{M}$  threads, data can be prefetched and computed without stalls inside each SM. This heterogeneous scheme can be executed efficiently on the GPU architecture as long as  $\mathcal{C}$  and  $\mathcal{M}$  threads are allocated into different *warps*. In such an implementation,  $\mathcal{C}$  threads never access slow GPU memory and can compute a larger number of stream graph executions using exclusively a small workset  $WS$  stored in SM memory. Concurrently,  $\mathcal{M}$  threads fetch the next input data from GPU memory to a double buffer  $DB$  in SM memory and store back the previous output data. A single synchronization point is required, when prefetched data from  $DB$  is swapped in  $WS$  and the previously computed results are swapped out from  $WS$  to  $DB$ .

The stream graph partitions supported by our implementation are connected through multiple input and output channels. Their corresponding data should be swapped between  $WS$  and  $DB$ . This is trivial only when a single input and output channel is involved [11]. In this case, the input channel corresponds to a contiguous range of memory locations, which overlaps with the output channel in  $DB$  and may also overlap in  $WS$ . Simply iterating through the data stored in  $DB$  in the correct direction ensures that no data is corrupted, and it is possible to swap data in parallel using multiple GPU threads. However, special care is required to support multiple channels.

The  $WS$  memory range corresponding to the channels of each graph operator is determined by a static memory allocator, based on liveness analysis. This allocator ensures that the channels receive a contiguous memory range (as a result, gaps may occur between channels). The input and output channels of a stream graph partition are also stored in  $WS$ , and the allocator may place them arbitrarily. If multiple channels need to be swapped from  $WS$  to  $DB$  and no additional constraints are in place, the actual location of the channels can lead to long dependency chains which may prevent swapping pairs of elements.

A possible scenario is shown in Figure 4a. The shaded boxes are the current channels in  $WS$  and  $DB$ . In this example the elements from input  $I_0$  can not be swapped into their designated location in  $WS$  as long as the contents of the output channel  $O_1$  has not been swapped out. However, this output channel can not be moved as it will corrupt  $I_1$  which has not been processed yet. Also,  $I_1$  can not be processed as it will corrupt  $O_0$ , etc. Utilizing temporary



**Figure 4.** Execution snapshot showing the challenges of partition I/O handling. The inputs for the next iteration have to swap with the outputs of the previous iteration.

memory storage is not feasible because the SM memory is limited and any extension degrades performance. Therefore, we propose an extension of the single channel swapping scheme that ensures that single element swaps can proceed without data corruption.

We direct the static allocator to layout the input channels without fragmentation from the first location in *WS*. This is possible as there are no previous data in *WS*. However, for outputs, we can not ensure that they are allocated in a contiguous fashion, but we can record the order in which they are allocated. The same order is replicated in the *DB*, where both input and output channels can be allocated contiguously. Such a layout is illustrated in Figure 4b.

Using this layout guarantees corruption free swapping, and we can prove this through induction on the index in *DB*. The basis case is for the first location in *DB*. The input stored at location 0 in *DB* can be moved to *WS*, and any output value it overwrites in *WS* can be moved to *DB* at the same location, because the outputs are compacted, in order, in *DB*. This can be implemented by storing the output first in a temporary register, and saving it afterwards to *DB*. If no output element exists in *WS* at location 0, there still obviously is no data corruption.

Assuming there is no data corruption until index  $p - 1$  in *DB*, when the input element at index  $p$  in *DB* is moved to location  $p$  in *WS* it may overwrite an unmoved output. In this case, the overwritten output element has to be moved to index  $q \leq p$  in the contiguous sequence of outputs in *DB*. This inequality is ensured by the contiguous allocation of input buffers in *WS* and *DB*, and the possible fragmentation of the outputs in *WS*. Therefore, the movement of the output to the index  $q$  in *DB* does not corrupt any input not yet transferred. This concludes the induction case if the number of inputs is larger than the number of outputs. Otherwise, the remaining outputs can be transferred to *DB* safely, as there is no remaining input in *DB*.

The automatically generated code relies on the above channel allocation. We infer a set of intervals where the swap indices for both input and output increase linearly. For each such interval, swaps can be applied to pairs of elements at consecutive locations.

### 6.3 Mapping Parameters Selection

Code generation for each GPU partition requires a few parameters, such as the number of  $C$  and  $M$  threads, or the number of parallel

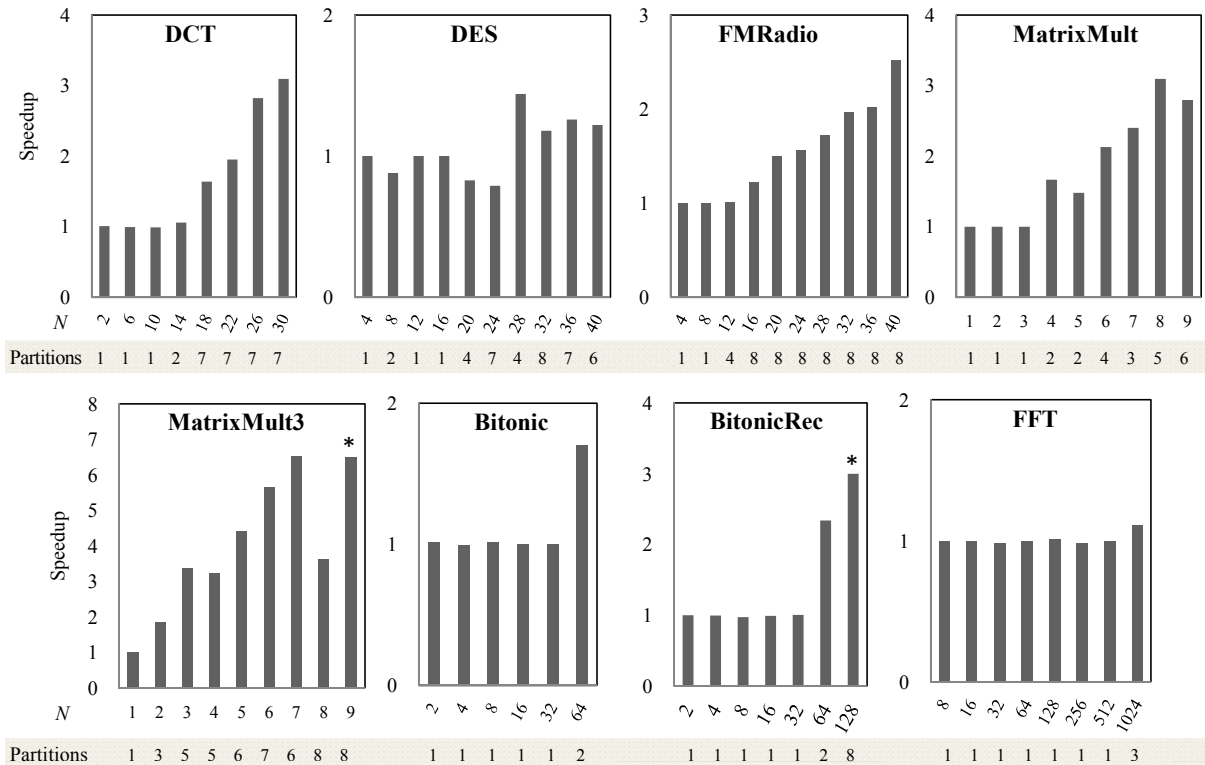
$C$  threads,  $S$ , supporting each stream graph partition execution. We first determine the number of concurrent executions of the partition that can be handled by each SM, based on the SM memory size and the memory requirement of each stream graph partition execution. Then, we allocate  $S$  threads for each partition execution, exploiting the data parallelism of the partition extracted through the stream graph structure. Finally, we need to match the data transfer requirements of the  $C$  threads with a corresponding number of  $M$  threads to minimize the stalls [11]. The same parameters are replicated for all SMs, as the SMs process parallel fragments of the input data. These parameters were estimated once during the performance evaluation of each partition in Section 5. However, during the final code generation step, we compute the exact SM memory layout, and the resulting footprint may increase due to fragmentation.

The number of  $N$  concurrent GPU streams utilized for the level 2 data transfer can influence how much of the CPU to GPU data transfer overhead is hidden. However, there is some penalty associated with each GPU partition kernel launch, and this surfaces if too many concurrent streams are utilized. In our implementation, we utilized 4 parallel streams to provide a good coverage of the memory transfer delays.

## 7. Experimental Results

In order to show the scalability and efficiency of our automated framework, we present the performance achieved on mapping several StreamIt benchmarks. These benchmarks (described in Table 1) are a representative set of the StreamIt benchmarks suite [2]. They were processed automatically by the framework, and code was generated for multiple partitions. The benchmarks were altered to create larger stream graphs by utilising a parameter  $N$  (i.e. the graph of DES for  $N = 40$  reached 1047 filters). The stream graphs are mapped onto one to four GPUs connected to the same CPU host. The benchmarks were augmented with source and sink filters that include code to verify the results of the computations. Because these filters maintain internal state, this also validated the support we provide for such filters. Our framework was implemented as a back-end to the StreamIt 2.1.1 compiler. The baseline CPU timing was obtained on an Intel Xeon CPU E5405 running at 2 GHz, with the executable generated through the uniprocessor back-end





**Figure 5.** Mapping to a single partition and to multiple partitions (the number of partitions is listed under the graphs) on a single GPU. The speedup is the execution time ratio between the two. Design points marked with (\*) were not supported by a single partition implementation.

of StreamIt, and compiled using the ‘-O3’ option of GCC 4.1.2. The experiments target the newer C2070 ‘Fermi’ GPU platforms.

**Comparison with the single partition mapping** Figure 5 shows the speedup achieved on a single GPU by our multiple partition mapping compared to the previous single partition mapping [11]. It also shows the number of partitions generated for each benchmark instance (the shadowed row under the values of  $N$ ). Most benchmarks benefit from multiple partitions when  $N$  increases. Using our proposed algorithm, multiple partitions yielded better performance than a single partition, because each partition requires a much smaller memory footprint. If only a single partition is used, the large working set resulted in poorer performance. To capture the CPU to GPU transfer overhead, the benchmarks maintain stateful source / sink filters. The additional speedup can be as high as  $6.53\times$ . In addition, there are a few cases where a single partition mapping could not return a solution (such as MatrixMult3 for size 9). However, for some benchmarks, such as Bitonic, DES or FFT, the working set size does not change significantly, and both single and multiple partitions mappings had similar performance.

**Multiple partitions on a single GPU** Figure 6 shows the speedup of the proposed partitioning approach relative to the CPU baseline. While the speedup may diminish for large values of  $N$ , this approach proves capable of sustaining good throughput for most benchmarks. If the size of the benchmark is too large to fit the SM memory, the benchmark is split into multiple partitions. In some cases, the overhead of data communication among the partitions severely impacted performance.

**Multi-GPU mapping** We tested large benchmarks running on multiple GPUs using the orchestration described in Section 6. The speedup obtained by running the benchmarks on 2 to 4 GPUs com-

pared to a single GPU mapping is shown in Figure 7. In general, when the size of the benchmark is not large enough, multiple GPUs do not provide any benefit. In these cases, a single GPU mapping is the best solution. The single GPU implementation corresponds to the white bars in the figure. This is mainly due to the communication overhead of transferring the data between the GPU and the CPU that could not be completely masked by computation.

However, the multi-GPU implementation proves profitable if  $N$  increases. As shown in Figure 6, the speedup of the single GPU mapping diminishes for large benchmarks. However, in this case, mapping to multiple GPUs starts to show its advantages. The speedup reaches  $2.97\times$  compared to a single GPU mapping. This is evidence that for applications that have large working sets, our multi-GPU solution can effectively speed up their execution.

Mapping to multiple GPUs (Figure 7) shows some divergent performance results for different values of  $N$ . The divergence can be explained because the nature of the stream graph itself may lead to solutions that are easily balanced on a specific number of GPUs, and adding additional GPUs may affect the balancing. Moreover, if significant communication exists between fine-grained filters, the performance will hardly increase if we put those filters across multiple GPUs.

Moreover, Figure 7 offers an indirect insight that the communication overhead can be effectively masked. The performance boost of a 2 GPU solution, compared to that achieved on a single GPU, is affected by several factors (such as how the workload is balanced between the 2 GPUs) in addition to the overhead of the complex communication mechanism. However, some design points of DES and MatrixMult3 mapped to 2 GPUs reach  $1.93\times$  and  $1.83\times$  speedup respectively, compared to a single GPU solution that does not have inter-GPU communication.



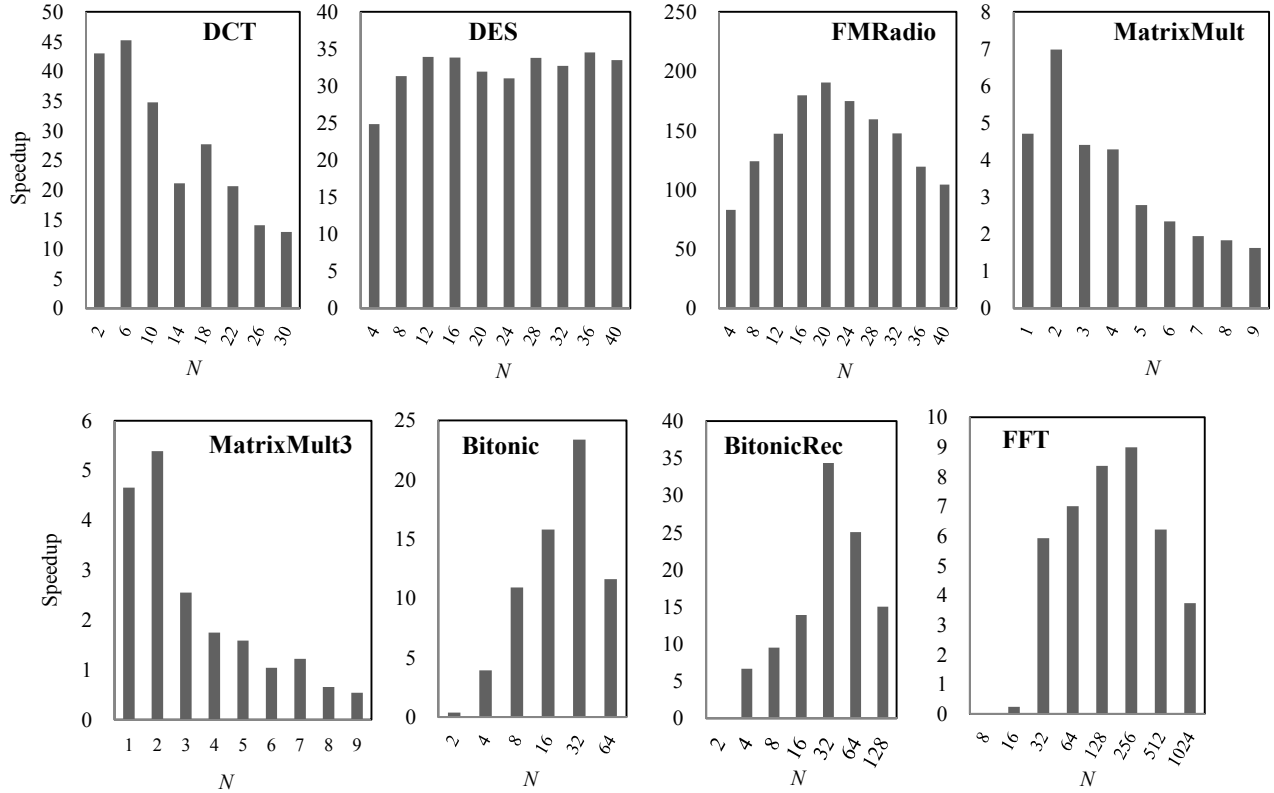


Figure 6. Mapping to a single GPU. The speedup is reported relative to a CPU implementation.

## 8. Conclusion

We proposed a scalable framework that automatically maps most stream processing applications onto GPUs. We developed an efficient graph partitioning algorithm that splits the complex application into several partitions, each of which can utilize the small on-chip SM memory effectively, and hence achieve good performance. Our proposed strategy obtained performance that augments that of a previous single partition solution. In addition, our proposed strategy is able to scale the performance to up to four GPUs. We also support stateful filters by running them on the CPU cores. The code generation scheme proposed is able to orchestrate the exchange of data within the individual GPUs, between the multiple GPUs, as well as the GPUs and the CPU cores. The results indicate the scalability and improvement when several GPUs are targeted.

It is conceivable that certain embarrassingly parallel applications can be mapped successfully to large scale multi-node, multi-GPU systems. However, on a single node, it is unlikely that the number of GPUs per node will increase significantly beyond the current four due to power, interconnect, and form-factor issues. Our work is the first to show that complex and often tightly coupled streaming applications can be successfully partitioned and mapped automatically onto multiple GPUs. As future work, we would like to investigate how even larger and more complex applications can be specified in StreamIt (or its derivative) so as to run on a large cluster of multi-GPU nodes.

## References

- [1] NVIDIA CUDA 4.0. <http://developer.nvidia.com/cuda-toolkit-40>.
- [2] Streamit benchmarks. <http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml>.

Benchmark	Description	Original $N$ [11]
Bitonic	Sorting algorithm for $N$ float elements applying the bitonic algorithm	8
BitonicRec	Same as above, recursive method	8
DCT	Discrete Cosine Transform for a matrix of $N \times N$ floats	8
DES	DES encryption algorithm with $N$ rounds, input 8 bytes, output as 16 hex digits	16
FFT	Fine grained FFT transform on $N$ elements	32
FMRadio	$(N + 3)$ -band equalizer radio	8
MatrixMult	Blocked matrix multiplication algorithm for $2N \times 2N$ matrices, split into blocks of $2 \times 2$	2
MatrixMult3	Same as above for $(3N + 3) \times (3N + 3)$ matrices, with blocks of $3 \times 3$	-

Table 1. Benchmark characterization.

- [3] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu. An adaptive performance modeling tool for GPU architectures. In *The 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*, 2010.
- [4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *ACM SIGGRAPH '04*, 2004.

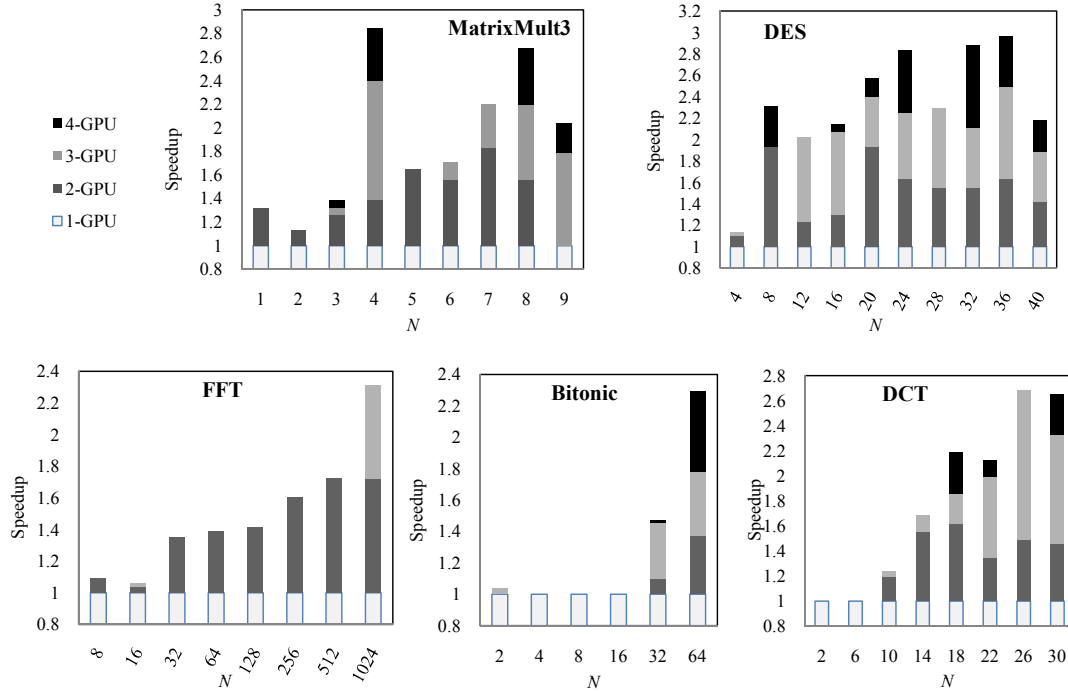


Figure 7. Additional speedup resulted from the mapping to multiple GPUs compared to a single GPU.

- [5] L. Chen, O. Villa, S. Krishnamoorthy, and G. R. Gao. Dynamic load balancing on single- and multi-GPU systems. In *2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, 2010.
- [6] G. Damos and S. Yalamanchili. Speculative execution on multi-GPU systems. In *2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, 2010.
- [7] C. M. Fiduccia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *The 19th Design Automation Conference (DAC '82)*, 1982.
- [8] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A stream compiler for communication-exposed architectures. In *The 10th international conference on Architectural support for programming languages and operating systems (ASPLOS '02)*, Oct 2002.
- [9] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *The 12th international conference on Architectural support for programming languages and operating systems (ASPLOS '06)*, 2006.
- [10] A. Hagiescu, W.-F. Wong, D. F. Bacon, and R. Rabbah. A computing origami: folding streams in FPGAs. In *The 46th Annual Design Automation Conference (DAC '09)*, 2009.
- [11] A. Hagiescu, H. P. Huynh, W. F. Wong, and R. S. M. Goh. Automated architecture-aware mapping of streaming applications onto GPUs. In *2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS '11)*, 2011.
- [12] A. H. Hormati, M. Samadi, M. Woh, T. Mudge, and S. Mahlke. Sponge: portable stream programming on graphics engines. In *The 16th international conference on Architectural support for programming languages and operating systems (ASPLOS '11)*, 2011.
- [13] H. P. Huynh, Y. Liang, and T. Mitra. Efficient custom instructions generation for system-level design. In *2010 International Conference on Field-Programmable Technology (FPT '10)*, 2010.
- [14] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 1998.
- [15] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 1970.
- [16] Khronos OpenCL Working Group. *The OpenCL Specification, version 1.0.29*, 8 December 2008.
- [17] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *The 2008 ACM SIGPLAN conference on Programming language design and implementation (PLDI '08)*, 2008.
- [18] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, 36(1), 1987.
- [19] J. Nickolls and W. J. Dally. The GPU computing era. *IEEE Micro*, 30, 2010. ISSN 0272-1732.
- [20] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1), 2007.
- [21] D. Schaa and D. Kaeli. Exploring the multiple-GPU design space. In *2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*, 2009.
- [22] J. A. Stuart and J. D. Owens. Multi-GPU MapReduce on GPU clusters. In *2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS '11)*, 2011.
- [23] A. Udupa, R. Govindarajan, and M. J. Thazhuthaveetil. Software pipelined execution of stream programs on GPUs. In *The 7th annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*, 2009.
- [24] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS '10)*, 2010.
- [25] Y. Zhang and J. D. Owens. A quantitative performance analysis model for GPU architectures. In *(The 17th International Symposium on High Performance Computer Architecture (HPCA '11))*, 2011.