Attribution–NonCommercial–NoDerivs 2.0 KOREA

You are free to :

- **Share** — copy and redistribute the material in any medium or format

Under the following terms :

Attribution — You must give [appropriate credit](#), provide a link to the license, and [indicate if changes were made](#). You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

NonCommercial — You may not use the material for [commercial purposes](#).

NoDerivs — If you [remix, transform, or build upon](#) the material, you may not distribute the modified material.

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

This is a human-readable summary of (and not a substitute for) the [license](#).

[Disclaimer](#)

COMMUNICATION-AWARE MAPPING OF STREAM GRAPHS FOR MULTI-GPU PLATFORMS

Dong Nguyen

Computer Engineering Program
Graduate School of UNIST

Communication-aware Mapping of Stream Graphs for Multi-GPU Platforms

A thesis
submitted to the Graduate School of UNIST
in partial fulfillment of the
requirements for the degree of
Master of Science

Dong Nguyen

11.20.2015
Approved by

Major Advisor
Jongeun Lee

Communication-aware Mapping of Stream Graphs for Multi-GPU Platforms

Dong Nguyen

This certifies that the thesis of Dong Nguyen is approved.

11.20.2015

Thesis Supervisor: Jongeun Lee

Woongki Baek: Thesis Committee Member #1

Wonki Jeong: Thesis Committee Member #2

Abstract

Stream graphs can provide a natural way to represent many applications in multimedia and DSP domains. Though the exposed parallelism of stream graphs makes it relatively easy to map them to GPGPUs, very large stream graphs as well as how to best exploit multi-GPU platforms to achieve scalable performance poses great challenges for stream graph mapping. Previous work considers either a single GPU only or is based on a crude heuristic that achieves a very low degree of workload balancing, and thus shows only limited scalability. In this paper we present a highly scalable GP-GPU mapping technique for large stream graphs with the following highlights: (1) an accurate GPU performance estimation model for (subsets of) stream graphs, (2) a novel partitioning heuristic exploiting stream graph's structural properties, and (3) ILP (Integer Linear Programming) formulation of the mapping problem. Our experimental results on a real GPU platform demonstrate that our techniques can generate significantly better performance than the current state of the art, in both single GPU and multi-GPU cases.

Contents

Contents	v
List of Figures	vi
I. INTRODUCTION	1
II. BACKGROUND AND RELATED WORK	3
2.1 Background	3
2.1.1 Input: Stream Graph	3
2.1.2 CUDA Programming Model	3
2.1.3 Single-GPU Mapping	4
2.2 Related Work	5
III. MAPPING STREAM GRAPHS TO MULTI-GPU	7
3.1 Our Mapping Flow	7
3.1.1 Strategy	8
3.1.2 Algorithm	9
3.2 Communication-aware Mapping	11
3.2.1 GPU Topology	12
3.2.2 ILP Formulation	13
3.2.3 Pipelined Multi-GPU Execution	16
3.3 GPU Performance Estimation Engine	16
3.3.1 Parameters and Profiling	17
3.3.2 Performance Model	17
IV. Experiments	19
4.0.1 Experimental Setup	19
4.0.2 Validation of Performance Model	20
4.0.3 Partitioning and Multi-GPU Mapping Results	21
4.0.4 Comparison with the Previous Work	22

4.0.5	The Validity and Accuracy of Our SOSP Metric	24
V.	Future Work	26
VI.	Conclusion	28
	References	29

List of Figures

2.1	Mapping a stream graph (a) to a single GPU. (b) One-kernel-per-filter approach. Thick blue arrows represent host interactions, and GM is GPU global memory. (c) One-kernel-for-graph approach. Compute threads (left box with a stream graph on it) and data transfer threads (center boxes with DT on them) run concurrently via double buffering, which requires intermittent buffer swapping.	4
3.1	Overall flow of the proposed multi-GPU mapping.	8
3.2	Shared memory size for (a) pipeline structure and (b) split structure.	9
3.3	GPU topology.	12
3.4	Creating a Partition Dependence Graph (right) from a stream graph (left). . . .	13
3.5	Three partitions running on a 2-GPU system. Each partition becomes a kernel. The communication from P2 to P3 is implemented as a pair of device-to-host and host-to-device data transfers, which are pipelined.	15
4.1	Accuracy of our performance estimation.	20
4.2	Scalability of our mapping technique. Speedup is over the 1-GPU, multi-partition mapping.	21
4.3	Multi-GPU performance, reported as the speedup over single-partition mapping. Results are shown for all applications whose results are reported in [7].	23
4.4	Four cases by two SW versions and two GPUs.	24
5.1	Shared memory map and filter operation of split structure. (a) an example of split structure. (b) original version. (c) enhanced version.	27

LIST OF FIGURES

5.2	Shared memory map and filter operation of join structure. (a) an example of join structure. (b) original version. (c) enhanced version.	27
-----	---	----

INTRODUCTION

Today GP-GPUs offer the highest performance per cost, making it the accelerator of choice for many applications. However, getting the maximal performance often requires ingenious programming effort as well as in-depth architecture knowledge [5]. One way to alleviate such difficulties is the use of a higher level of abstraction such as stream graphs. A stream graph consists of filters (also called actors) and channels representing data flow between them, and can provide a more intuitive and natural representation for stream processing type of applications such as multimedia and DSP.

In the literature there are two approaches of mapping a stream graph to a GPU. The first approach [5] creates one kernel for one filter, which is straightforward but requires a heavy use of the GPU global memory for inter-filter communication. To avoid the global memory bottleneck problem, the second approach [4] creates one kernel out of the entire stream graph, implementing inter-filter communication via on-chip scratchpad memory called *Shared Memory (SM)*, which is very efficient. However, the limited size of the SM mandates that only a limited number of executions may run concurrently per SM and there have to be dedicated *data transfer* threads running concurrently with *compute* threads in order to best utilize the SM. Thus while the second approach generates higher performance in general, achieving optimal mapping requires careful balancing between compute vs. data transfer threads in addition to determining other key parameters.

The problem of balancing compute vs. data transfer takes on a new level of difficulty when

it comes to mapping stream graphs to a multi-GPU platform. This is because we now need to balance workload among multiple GPUs, and also because communication between GPUs may not always be hidden by computation time. Sometimes communication may be the performance bottleneck determining the application throughput. Thus we need to find an accurate way of estimating the computation and communication time, so as to find a mapping that maximizes the overall throughput. Another challenge with multi-GPU mapping is graph partitioning. Finding the right set of partitions is important, since it makes a permanent impact on the ensuing mapping step. It is also very challenging because during the partitioning step it is hard to foresee how the partitioning decision will affect the quality of mapping in the mapping step.

In this paper we present a novel partitioning-based GPU mapping methodology for stream graphs, featuring the following contributions.

- An accurate GPU performance model for stream graphs
- A novel partitioning heuristic exploiting stream graph's structural properties
- An ILP (Integer Linear Programming) formulation for multi-GPU mapping

Our experimental results on a real hardware platform demonstrate that our technique can generate scalable performance for up to 4 GPUs, achieving 3.2 times speedup on average for large stream graphs compared to 1-GPU mapping, and can generate mappings that are far better, as measured in the speedup over single-partition mapping, than the previous work [7] for compute-bound applications.

The rest of this thesis is organized as follows. After reviewing the background and some related works in Chapter II, we describe our multi-GPU mapping in Chapter III. The experimental results are presented in Chapter IV. Then we present our future work related to some mapping optimization and show some preliminary results in Chapter V. Finally we conclude the paper in Chapter VI.

BACKGROUND AND RELATED WORK

2.1 Background

2.1.1 Input: Stream Graph

Stream processing is a model of computation for applications where input and output are streams of data such as video and audio signals. There are several languages designed for stream processing, but in this paper we use StreamIt [9], which supports concise representation of stream processing applications through a small set of composition operators such as *pipeline* (consecutive filters), *split-join* (fan-out), and *feedback loop* (cyclic structure).

2.1.2 CUDA Programming Model

CUDA is a multi-thread parallel computing model for Nvidia GPUs. It defines two kind of devices: the host (CPU) which usually executes sequential code while the massive parallel code portion (kernel) is mapped on the device (GPU). At the hardware level, a GPU consists of a number of streaming multiprocessors, all of which run in parallel. Each streaming multiprocessor contains many streaming processors (SPs), which are simple processor cores.

CUDA threading model is organized to threads, blocks and grids. At the highest level, all threads that execute the same kernel form a grid. A grid is a 1 or 2 dimensional array that consists of many thread blocks. A thread block is a collection of threads running on multiprocessor

Figure 2.1: Mapping a stream graph (a) to a single GPU. (b) One-kernel-per-filter approach. Thick blue arrows represent host interactions, and GM is GPU global memory. (c) One-kernel-for-graph approach. Compute threads (left box with a stream graph on it) and data transfer threads (center boxes with *DT* on them) run concurrently via double buffering, which requires intermittent buffer swapping.

at a given time and can be up to 3 dimensional array. A thread, which is the basic executing unit is map to a single processor core. All threads in a block are assigned to a streaming multiprocessor and execute in a SIMT (Single Instruction, Multiple Thread) fashion. Threads are divided into groups of 32 threads referred to as warps. Threads in a warps execute the same instruction at the same time unless they encounter a divergence in the code path.

CUDA memory model consists of 4 levels of hierarchy. The lowest level is registers, which is on chip and per thread memory. Each thread also has private access to local memory which locates in off-chip DRAM. All threads in the same block can access to an on chip memory referred as Shared Memory (SM). SM has low latency as comparable to register but is size-limited. The next level of memory is global memory which is visible to all threads in the grid. The latency of this kind of memory is about 400–600 times compared to that of SM [10].

2.1.3 Single-GPU Mapping

As mentioned earlier there are largely two approaches to mapping a stream graph to a GPU. In the more straightforward approach illustrated in Figure 2.1(b), each filter is converted into a GPU kernel. Since stream graphs are executed many times, each kernel (which is equivalent to a filter in this approach) has many iterations or threads, needed for efficient GPU execution. However the large number of iterations per filter also means that each filter will produce a large amount of data that must be consumed by its dependent filters, making it necessary to utilize the GPU global memory for inter-filter communication. The heavy use of the off-chip global

memory increases the communication-to-computation ratio, making global memory access easily the performance bottleneck.

To avoid the global memory bottleneck problem the second approach creates a single GPU kernel out of the entire stream graph as illustrated in Figure 2.1(c). Then the inter-filter communication can be implemented through the SM which is very efficient. The global memory access is still needed for primary input/output data, which can be done by a dedicated group of threads, called *data transfer* threads. To hide global memory access latency, it is essential to run data transfer threads concurrently with *compute* threads via double buffering. The existence of two types of threads and the cap on the total number of threads per SM as well as the limited SM size create an interesting optimization problem, of determining the number of compute vs. data transfer threads. To make it even more complicated, using multiple compute threads per execution of a stream graph can result in higher performance if filters have greater-than-one firing rates, which is often the case (firing rate is the number of times a filter has to be executed in order to match its input and output rates). All these parameters—viz., the number of threads per execution, the number of executions per SM, and the number of data transfer threads—need to be determined simultaneously for optimal results, further adding to the challenge. Our technique is based on the second approach, but significantly extends it addressing the new challenges of multi-GPU mapping.

2.2 Related Work

This work is built on top of previous work [7], which also aims at scalable stream graph mapping for multi-GPU platforms. Our work shares with it the same single-GPU mapping approach, which was first proposed in [4], the same multi-GPU execution model and the overall framework of partitioning-followed-by-mapping. However, our scheme significantly improves the previous work, by introducing a series of new techniques: an accurate GPU performance model, a novel partitioning heuristic, and an ILP-based optimal mapping scheme that explicitly considers inter-GPU communication overhead. All these techniques work together to generate significantly higher performance over the previous work as demonstrated by our experimental results.

There are several approaches to mapping stream graphs to GPUs [4, 5, 7, 10] or GPU clusters [11]. Udupa et al. [10] also use ILP formulation to map StreamIt programs to a GPU. Hormati et al. [5] proposes a set of compiler optimizations to increase the efficiency of programs written in StreamIt. Zhang et al. [11] proposes a slightly different stream processing language and associated tools targeted at GPU clusters.

Stream graphs have been mapped to other architectures as well, including the Raw processor [2], the STI Cell processor [1, 8] and FPGAs [3]. Those techniques often focus on exploiting task-level parallelism and balancing workloads among multiple processors. To make it easier to balance workload among multiple processors, fusioning/fissioning of stateless filters is used [3, 8].

Integer Linear Programming (ILP) has been used for mapping stream graphs [1, 8, 10]. Though some of them also target multi-core architectures, none of them explicitly model and optimize to minimize the impact of inter-processor communication.

CHAPTER III

MAPPING STREAM GRAPHS TO MULTI-GPU

3.1 Our Mapping Flow

Figure 3.1 illustrates the main steps of our mapping flow. The input to our framework is a stream graph whose nodes are annotated with GPU execution time as explained in Section 3.3.1. A stream graph is a directed graph with nodes representing filters (one-to-one correspondence between nodes and filters) and edges representing the data flow between nodes. Associated with a node is the firing rate of the filter. An edge has a weight corresponding to its buffer size. The main steps of our flow are graph partitioning and multi-GPU mapping. The partitioning step is aided by the performance estimation engine, which can provide estimated GPU execution time for any given subgraph of a stream graph. The performance estimation engine also finds out the parameters needed to achieve the estimated performance. After partitioning, the multi-GPU mapping step determines partition-to-GPU mapping considering both workload and communication balancing. We use the ILP (Integer Linear Programming) formulation, which can find the optimal solution that can minimize the expected runtime. Finally the mapping result is passed to the GPU code generator, which generates the CUDA code that can run on real GPU hardware. It also generates additional code necessary for kernel launch, data transfer handling, synchronization, and pipelined execution of multiple GPUs.

Figure 3.1: Overall flow of the proposed multi-GPU mapping.

3.1.1 Strategy

Finding the best partition of a stream graph for multi-GPU mapping is a very hard problem. Even the optimal number of partitions is not known,¹ necessitating heuristic approaches. Since the single most important factor impacting GPU performance is the SM utilization, previous work [7] uses a partitioning heuristic that keeps merging filters until the SM requirement is violated.

We go one step further and optimize explicitly to minimize the total runtime of partitions as predicted by our performance estimation engine. Even with this goal, the large solution space of partitioning demands a heuristic for efficient search.

Our algorithm employs the following strategies. First we generate partitions that use less SM by exploiting the structure of stream graphs. Second, partitions are merged only if doing so is expected to reduce the total runtime. Third we minimize the number of partitions that are IO-bound as opposed to compute-bound. Lastly for easier workload balancing during mapping, we make partitions similar-sized as measured in workload.

Regarding the first strategy, our heuristic exploits the fact that stream graphs are composed using pipeline and split/join operators. If a pipeline operator is used, the SM requirement of the pipeline is not much higher than that of the filters comprising the pipeline, as illustrated in Figure 3.2(a). This is because those filters usually have identical push/pop rates and short-lived buffers. In contrast, a split/join structure often has high SM requirement because their buffers have longer life times, as illustrated in Figure 3.2(b).

¹It is not necessarily equal to the number of GPUs, since a large partition may better, in terms of execution time, be partitioned and run sequentially.

Figure 3.2: Shared memory size for (a) pipeline structure and (b) split structure.

To realize the second strategy, we use function $T(p)$ that generates an execution time estimate for any given partition p , or a subgraph of a stream graph. As for the third strategy, we regard a partition p as compute-bound (or IO-bound) if its compute time estimate (i.e., $T_{comp}(p)$) is greater (or less, respectively) than its data transfer time estimate (i.e., $T_{dt}(p)$). The two terms, $T_{comp}(p)$ and $T_{dt}(p)$, are also provided by the performance estimation engine, as explained in Section 3.3.

3.1.2 Algorithm

Our partitioning algorithm works in four phases. The first phase merges filters within pipelines, the second phase merges those outside pipelines, and the third phase merges the two types of partitions together. The final phase attempts simultaneous merging of two neighboring partitions and of all the filters.

Algorithm 1 illustrates our partitioning algorithm. Here *Try-Merge* represents a conditional merge operation between two partitions or between a partition and a node, which can happen only if (i) they are connected, (ii) the merged partition is convex,¹ and (iii) the execution time of the merged is (expected to be) smaller than the combined execution time of each. The last condition implies that the merged partition should not violate the SM size constraint.

¹A partition is *convex* if there is no path between two internal nodes that goes through an external node [7].

Algorithm 1 Partition-Stream-Graph

```

1: // Phase 1
2: for all pipeline pipe do
3:    $h \leftarrow$  the first node of pipe
4:   repeat
5:     new partition  $p \leftarrow \{h\}$ 
6:     repeat Try-Merge one neighbor node in pipe with  $p$ 
7:       until no merging occurs or no more neighbor node
8:      $h \leftarrow$  last failed node
9:   until no more unassigned node in pipe
10: end for
11:
12: // Phase 2
13: for all node  $n$  not belonging to any partition do
14:   new partition  $p \leftarrow \{n\}$ 
15:   repeat
16:     for all neighbor node  $k$  of  $p$  not in any partition do
17:       Try-Merge  $p$  with  $k$ 
18:     end for
19:   until no merging occurs
20: end for
21:
22: // Phase 3: this phase is repeated three times (see text)
23:  $L1 \leftarrow$  list of all IO-bound partitions
24:  $L2 \leftarrow$  list of all compute-bound partitions
25: repeat
26:   for all  $p \in L1$  in ascending order of execution time do
27:     Try-Merge  $p$  with another (connected) partition in  $L1$ 
28:     if merging succeeds, break
29:   end for
30:   update  $L1$  and  $L2$ 
31: until no merging occurs
32:
33: // Phase 4
34: Perform simultaneous merging of two neighboring partitions
35: Perform simultaneous merging of all the nodes
  
```

The first phase generates partitions within pipelines (lines 2–10). First, all innermost pipelines are identified. Within each pipeline, we find maximal connected subsets that can be merged. This is done by initially creating a partition that includes only the first element of the pipeline, and repeatedly expanding it to include the neighboring node until it no longer satisfies the merging criteria. At that point, if there remains a node not belonging to any partition, we start a new partition and repeat the same procedure.

The second phase (lines 13–20) generates partitions outside innermost pipelines, or among the nodes that belong to no partition yet. We simply keep merging the nodes until no merging can occur.

The third phase (lines 23–31) merges the partitions generated so far. At this point every node belongs to a partition. To steer the partitions to become as much compute-bound as possible, we prioritize mergings (i) between IO-bound ones, then those (ii) between an IO-bound and a compute-bound one, and those (iii) between compute-bound ones, in that order. This is because merging two partitions often reduces the total data transfer time (thereby making IO-bound ones compute-bound), thanks to the shared buffer between them.¹ Also to balance partition sizes, we give priority to those partitions that are smaller in workload.

To implement this we create two lists. $L1$ is the list of IO-bound partitions, and $L2$ is that of compute-bound ones. This phase is repeated three times: first within $L1$ only, then between $L1$ and $L2$ ($L1$ in line 27 is replaced with $L1 \cup L2$), and for both $L1$ and $L2$ ($L1$ is replaced with $L1 \cup L2$ in both lines 26 and 27).

The last phase attempts to merge (1) two neighboring partitions at once, which may reduce the runtime even though merging either of them does not, and (2) all the nodes at once, which can help guarantee that our multi-partition solution is no worse than the single-partition solution.

3.2 Communication-aware Mapping

Compared to single-GPU mapping, mapping to a multi-GPU target generally incurs higher inter-processor communication while the workload per GPU becomes less. As a result communication can more easily be a performance bottleneck especially for IO-intensive applications. On the other hand, if communication is not a bottleneck, further reducing the amount of communication does not improve performance at all. This suggests that for optimal multi-GPU mapping we need to accurately model inter-GPU communication time as well as GPU computation time, which is not attempted in the previous work.

¹At the same time merging may increase the compute time as compared with the combined compute time of each, due to the limited number of threads available per kernel.

Figure 3.3: GPU topology.

GPU computation time can be easily obtained using our performance estimation engine. However modeling inter-GPU communication time in a way that can be used in an ILP (Integer Linear Programming) formulation is challenging.

3.2.1 GPU Topology

Part of the complication arises because GPUs are not symmetrically placed to each other. Figure 3.3 illustrates the GPU topology of a 4-GPU machine, which is a tree with GPUs as leaves and PCI Express switches as internal nodes. The edges are full-duplex PCI Express links. When using peer-to-peer communication, data first traverse uplink to the lowest common ancestor, and then traverse downlink to the destination node. As a result, for instance, if GPU 1 sends data to GPU 2, only two links are used, but if GPU 2 sends data to GPU 3, four links are used.

We model the communication time at a link to be a linear function of the total load on the link, direction-wise. The total load of a link is the sum of all communication data passing the link in the direction considered. The link with the highest load becomes the communication bottleneck, and determines the inter-GPU communication time in our mapping.

Thus we need to find out the total load on a link, which is easy if we have the partition-to-GPU mapping information, but finding it out (in an algebraic form) without the mapping information is challenging.

We observe that for any given link, it is possible to determine the source-destination pairs whose communication will contribute to the load of the link. For example, the link $SW2 \rightarrow SW1$ in the figure will be used only by the communication between these GPUs: (1, 3), (1, 4), (2, 3), and (2, 4), where the first number is the source GPU ID and the second number the destination. In fact, for a tree topology we can derive a simple rule, which we use in our implementation:

Figure 3.4: Creating a Partition Dependence Graph (right) from a stream graph (left).

the load of an uplink l is contributed by the data transfer from GPU i to GPU j if and only if i is a child of l and j is not. (A GPU is a child of a link if it is a child of either end point of the link.) Clearly, for any general topology we can make the list of communication source-destination pairs contributing to each link's load, which can be used in our ILP formulation. Though this approach can create a large number of variables and constraints, the complexity (i.e., the number of constraints) grows only quadratically with the number of GPUs (see (III.6)).

3.2.2 ILP Formulation

Input: *Partition Dependence Graph (PDG)* is a directed graph (V_P, E_P) , where $V_P = \{p_i\}$ is the set of partitions generated from the partitioning step, and E_P is the set of edges between partitions. Let P be the number of partitions. Associated with each partition p_i is the workload information T_i , which is the value of $T(p_i)$ as reported by the performance estimation engine. An edge (p_i, p_j) exists if the stream graph has at least one edge from any node of p_i to any node of p_j , and its weight is D_{ij} , which is the sum of all the weights of the edges connecting a node of p_i to that of p_j (see Figure 3.4). Also given are the number of GPUs, denoted by G , and the GPU topology, which has L uni-directional links. We assume that GPUs are homogeneous, but our ILP formulation can also be extended to heterogeneous cases.

Objective: Minimize T_{max}

T_{max} is the greatest execution time on any GPU or any communication link, and determines the application throughput of the mapping.

$$T_j^{gpu} \leq T_{max} \quad \forall j \in [1, G] \quad (\text{III.1})$$

$$T_l^{comm} \leq T_{max} \quad \forall l \in [1, L] \quad (\text{III.2})$$

$$T_l^{comm} = Lat + D_l/BW \quad \forall l \in [1, L] \quad (\text{III.3})$$

Here Lat is the initial latency of communication, and BW is the bandwidth of a link.

GPU Time: Using binary decision variable n_{ij} , which is 1 if p_i is mapped to GPU j , and 0 otherwise, GPU time on GPU j can be represented as follows.

$$T_j^{gpu} = \sum_{i=1}^P n_{ij} \cdot T_i \quad \forall j \in [1, G] \quad (\text{III.4})$$

Every partition must be mapped to one GPU.

$$\sum_{j=1}^G n_{ij} = 1 \quad \forall i \in [1, P] \quad (\text{III.5})$$

Inter-GPU Communication: For every edge (i, j) of PDG, we introduce binary variable e_{ijkh} indicating that p_i and p_j are mapped to GPU k and h , respectively.

$$e_{ijkh} = n_{ik} \cdot n_{jh} \quad \forall (i, j) \in E_P, \quad \forall k, h \in [1, G] \quad (\text{III.6})$$

The above is not linear, but can be easily linearized because all the variables are binary. Then the communication load D_l on link l , which is the total amount of data transfer passing the link, can be represented as follows.

$$D_l = \sum_{(i,j) \in E_P, (k,h) \in dtlist(l)} e_{ijkh} \cdot D_{ij} \quad \forall l \in [1, L] \quad (\text{III.7})$$

Here $dtlist(l)$ the list of source-destination GPU pairs, whose communication data will go through the link l .

Figure 3.5: Three partitions running on a 2-GPU system. Each partition becomes a kernel. The communication from P2 to P3 is implemented as a pair of device-to-host and host-to-device data transfers, which are pipelined.

3.2.3 Pipelined Multi-GPU Execution

After partition-to-GPU mapping, each partition is converted to a GPU kernel using the second approach of Section 2.1.3. Partitions mapped to the same GPU run sequentially as illustrated in Figure 3.5, where P1, P2, and P3 represent partitions.

To support the pipelined execution of multiple partitions on different GPUs, the input data stream is divided into N fragments [7]. For each GPU, N asynchronous streams are generated corresponding to N data fragments, one at a time in a pipelined manner. Each stream coordinates data transfer (i.e., device to host or host to device) and kernel execution (for one or more kernels) for its corresponding fragment. As the operations on these fragments are independent, memory transfers and kernel executions of different fragments can overlap, forming a pipeline, so that the inter-GPU communication latency can be hidden. For example, when GPU 1 is performing kernel execution of partitions P1 and P2 for fragment n , the data of fragment $(n - 1)$ is being transferred to host. At the same time GPU 2 receives the data of fragment $(n - 2)$ from host and performs kernel execution of P3 corresponding to data fragment $(n - 3)$.

One difference from [7] is the use of peer-to-peer communication, which is more efficient than having every inter-GPU communication go through CPU, as is done in the previous work.

3.3 GPU Performance Estimation Engine

Our Performance Estimation Engine (PEE) provides GPU execution time estimate for any subgraph of a stream graph. Such a performance prediction can be extremely valuable in making partitioning decisions and performing communication-aware mapping. The main challenge is how to accurately predict the subgraph-level performance from node-level performance information only that is provided by initial profiling. In order to have high accuracy we find these two things essential: minimization of static discrepancy, and a good performance model.

Static discrepancy can be caused by the mismatch between the PEE and the GPU code generator in terms of how to convert a subgraph into a GPU kernel. Creating a GPU kernel requires determining several key parameters including the number of compute / data transfer threads and the number of executions per kernel. Even the same kernel code may show very different performance if, for instance, a different number of threads are assigned. To minimize the static discrepancy the PEE includes the same optimization done by the GPU code generator, and saves the parameters found to be optimal for later GPU code generation. The fact that in our multi-GPU mapping flow, each subgraph, if selected as a partition, will eventually become a kernel by itself, also helps minimize the static discrepancy.

After parameter selection, the PEE uses a GPU performance model to statically estimate the execution time of a GPU kernel. Static estimation is essential due to the large number of GPU kernels to evaluate.

Let us now explain our GPU performance model after describing major parameters obtained from profiling and kernel code optimization.

3.3.1 Parameters and Profiling

Given a stream graph, we first annotate each node with its GPU execution time t_i . GPU execution time is obtained from simulation after converting each filter to a GPU kernel with data prefetching suppressed. This is to find the time spent on just filter computation. We run each kernel assigning a single GPU thread.

Other parameters come from kernel code optimization. Since we take the second approach of Section 2.1.3, which is illustrated in Figure 2.1(c), there are a number of *compute* threads and *data transfer* threads. Let F be the number of data transfer threads. The number of compute threads is given as $W \cdot S$, where W is the number of executions per kernel, or the number of loop iterations executed together, and S is the number of compute threads per execution.

3.3.2 Performance Model

We first develop our performance metrics for a kernel with W executions, and later normalize it to allow comparisons between different-size partitions.

Total Execution Time: From Figure 2.1(c), the total execution time of a kernel, T_{exec} , can be represented as follows.

$$T_{exec} = \max(T_{comp}, T_{dt}) + T_{db} \quad (\text{III.8})$$

Here T_{comp} is the time that compute threads spend on the filter's computation. Compute threads access the SM, but do not access the global memory. T_{dt} represents the time for data transfer threads to transfer data between the SM and the global memory. Since compute threads and data transfer threads are assigned to distinct warps, the total execution time is determined by the maximum of T_{comp} and T_{dt} . After compute threads and data transfer threads are all finished, the working set (WS) buffer and the double buffer (DB) need to be swapped, the time for which is represented as T_{db} .

Compute Time: Compute time of a partition would be simply the sum of t_i of its nodes if only one thread is used per execution. Taking into account the S factor (the number of compute threads per execution) and that some filters may not have enough firing rate f_i to fully utilize all

the S threads, we can see that each node of a partition will utilize $\min(f_i, S)$ compute threads only, giving the following formula for compute time of W executions.

$$T_{comp} = \sum_i \frac{t_i}{\min(f_i, S)} \quad (\text{III.9})$$

Data Transfer Time: Through experiments we observe that data transfer time, of a kernel with W executions, is linearly proportional to the amount of IO data size D and inversely proportional to the number of data transfer threads F . Also all threads are involved in WS and DB swapping. The amount of swapped data is the same as the IO data size.

$$T_{dt} = C_1 \frac{D}{F} \quad (\text{III.10})$$

$$T_{db} = C_2 \frac{D}{F + W \cdot S} \quad (\text{III.11})$$

Here C_1 and C_2 are design parameters that are empirically determined.

Execution Time: Finally we define the execution time of a partition as follows. This is to allow a direct comparison between partitions of different sizes.

$$T = \frac{T_{exec}}{W} \quad (\text{III.12})$$

CHAPTER IV

Experiments

4.0.1 Experimental Setup

We have implemented our mapping flow as a back-end to the StreamIt compiler. The StreamIt compiler generates C code with each filter as a separate function, from which our tool reconstructs a stream graph, then performs profiling (which uses GPU code generation), partitioning, ILP-based mapping, and the final GPU code generation. The multi-GPU mapping step took no more than 10 seconds at most with a modern ILP solver [6]. For the performance model in the PEE, we use $C_1 = 38.4$ and $C_2 = 11.2$, which are empirically found from a linear regression of the profiled data (see Section 4.0.2). Finally our mapping flow emits CUDA code, which is then compiled using Nvidia *nvcc* 6.0 and run on a Xeon workstation with 4 Nvidia M2090 GPUs.

We evaluate our proposed technique using stream applications from the StreamIt distribution [9]. We use all the eight applications evaluated in the previous work [7], where one can also find the description of the applications and the size parameter N .

We present three sets of results: (i) validation of our GPU performance estimation, (ii) our multi-GPU mapping results, and (iii) the comparison with the previous work [7]. All the performance numbers reported here for our technique are real measurements on the mentioned system.

Figure 4.1: Accuracy of our performance estimation.

4.0.2 Validation of Performance Model

To see the accuracy of our performance model we use all the partitions finally selected and passed to the mapping stage. There are about 350 unique partitions, for which performance predictions are made. Then after the partitions are mapped to GPUs and CUDA kernels are generated (one partition corresponds to one kernel), we compile and run them on the 4-GPU machine, and collect the actual runtime information of the CUDA kernels using the Nvidia profiler. (Only the kernel execution time is included, excluding kernel launch time.) Thus in actual measurement all the kernels of an application run together, whereas our performance estimation engine considers each partition in isolation.

Figure 4.1 compares our predictions with actual measurements, which suggests that there is a strong correlation between the two, with the R-squared value of 0.972. One important factor contributing to this high correlation is that each node of a stream graph has very little dynamism in terms of execution time. In particular a node’s execution time is typically invariant to the input value.

In the graph we observe that in most cases the difference is insignificant, but some data points have severe deviation where the actual runtimes are typically higher than our predictions. We believe this is due to the SM bank conflict between compute and data transfer threads. All

Figure 4.2: Scalability of our mapping technique. Speedup is over the 1-GPU, multi-partition mapping.

in all, for most data points our prediction is accurate, and significant deviations occur very infrequently.

4.0.3 Partitioning and Multi-GPU Mapping Results

To evaluate the effectiveness of our communication-aware mapping, we map each application to different numbers of GPUs. For each application and parameter N , our partitioning heuristic generates one set of partitions, which is mapped to 1 through 4 GPUs. Other than the number of GPUs, everything else is the same among the four cases. Figure 4.2 shows the results.

Partitioning results: First the number of partitions generated of each application for different N values is shown on the x-axis of Figure 4.2.¹ These numbers are almost always greater than or equal to that of the previous work [7],² which is not surprising since we use a stricter set of criteria for merging.

¹Due to the stochastic nature of our partitioning heuristic, fewer partitions may be generated from a larger graph, as is the case with *FMRadio*.

²[7] targets a slightly different GPU, Nvidia C2070. However, it has the same SM size and Compute Capability as our target GPU, and since the previous work's partitioning criteria are solely based on SM requirement, using the new GPU should not affect their partitioning results significantly.

Second, what is interesting is that the increase in the number of partitions is not uniform among the applications. In some applications the number is increased by more than 10 times while in some others it is nearly the same. The average is about 3.7 times (geometric mean).

Third, if we define *kernel count ratio* as the number of partitions in ours vs. that of the previous work, those with high kernel count ratios tend to be compute-bound. This can be explained as follows. In compute-intensive applications such as *DES* and *DCT*, partitions reach the state of compute-boundedness more quickly. Since compute time does not decrease by partition merging—rather, it tends to increase because a limited number of threads must be shared by the partitions—compute-bound partitions are less likely to get merged under our merging criteria, resulting in higher number of partitions.

Scalability: In Figure 4.2 applications are listed in the decreasing order of kernel count ratio, with 5 compute-bound (kernel count ratio is 3 or more) and 3 memory-bound (1.5 or less). First we note that the speedup for the largest N is quite high. In 5 out of 8 applications, we achieve nearly $3.5\times$ or higher speedup with 4 GPUs; in one application, *DCT*, the final speedup is close to $3\times$; and the other 2 applications stop at $2.5\times$. This is a marked improvement over the previous work,¹ and demonstrates the scalability of our communication-aware mapping scheme.

Second, most applications show progressively increasing speedup as N increases. When N is small, the amount of computation is small, and dividing the workload among multiple GPUs may not give enough benefit compared to the communication cost. As N increases, workload also increases, and utilizing more GPUs can give enough benefit to compensate for the communication cost. Third, as a consequence, compute-bound applications tend to benefit more quickly from the use of multiple GPUs, though half the applications fall in the middle class.

Overall, as the number of GPUs is increased to 2, 3, and 4, our ILP-based mapping scheme can achieve the final speedup, or the speedup for the largest N , of $1.8\times$, $2.6\times$, and $3.2\times$ on average across all the applications, compared with 1-GPU multi-partition mapping, reinforcing the need for multi-GPU mapping algorithms such as ours.

4.0.4 Comparison with the Previous Work

Finally we provide a quantitative comparison between our multi-GPU mapping and the previous work [7].² For a fairer comparison we use a target architecture that is very similar to

¹Direct comparison of ours with [7] in terms of scalability has some issues including different baselines and different numbers of applications evaluated. But at least our 1-GPU performance is not necessarily worse than that of the previous work, as shown in Figure 4.3.

²Multiple issues in the paper have prevented us from fully reproducing [7]; hence, we have chosen to cite the numbers in the paper instead.

Figure 4.3: Multi-GPU performance, reported as the speedup over single-partition mapping. Results are shown for all applications whose results are reported in [7].

that of the previous work, including the same CUDA capability and the same SM size. Still, other differences prevent us from directly comparing the raw performance between the two GPUs. Thus we use a relative speedup factor called *Speedup Over Single-Partition mapping (SOSP)*, which is defined as the *relative performance* of a multi-partition multi-GPU mapping to its *Single-Partition Single-GPU (SPSG)* mapping *on the same hardware*. Though not perfect, the SOSP metric is useful because both ours and the previous work implement the same SPSG heuristic of [10]. Also the SOSP metric will be far less sensitive to small hardware changes than raw performance numbers.

Figure 4.3 compares our proposed scheme vs. the previous work in terms of SOSP for 1 to 4 GPUs. Results are shown for all five applications whose multi-GPU mapping performance is reported in the previous work. For compute-bound applications, our multi-GPU mapping produces solutions that far outperform those of the previous work regardless of the parameter N . This is the combined effect of our partitioning heuristic and the multi-GPU mapping, and demonstrates the effectiveness of our communication-aware mapping. For memory-bound applications we see mixed results, with one exceptional case of *MatMul3* at $N = 8$, where our partitioning heuristic returns just one partition. All in all our technique can generate mappings that are on average 17%, 33%, 40%, and 47% better *in terms of SOSP* than those of the pre-

Figure 4.4: Four cases by two SW versions and two GPUs.

vious work for 1-, 2-, 3-, and 4-GPU cases, respectively. For compute-bound applications the improvements are even higher. Note that we do *not* claim these numbers as measured performance difference on the same GPU, but we provide them as our best estimation of the expected performance differences between the two algorithms.

4.0.5 The Validity and Accuracy of Our SOSp Metric

Let us consider four combinations produced by two software versions, SPSG (Single-Partition Single-GPU-mapped) code and MPMG (Multi-Partition Multi-GPU-mapped) code, and two slightly different GPUs of the same architecture, G1 (C2070) and G2 (M2090), as illustrated in Figure 4.4.

First, the multi-GPU code generated by the previous work is identical regardless of the target GPU as long as the SM size is the same. This is because (i) their partitioning heuristic considers the SM size only, without regard to any other characteristics such as clock speed or memory bandwidth, and (ii) their multi-GPU mapping method is hardware-agnostic. Therefore the SW code in Figure 4.4(d) is exactly the same as that of (b). Second, since the SW is the same, and G2 is only a scaled-up version of G1 with the exactly same architecture—the only differences being (i) GPU core clock, (ii) GPU memory clock, and (iii) stream multiprocessor count—we can expect the performance difference between Cases (d) and (b) to be very similar to that of the GPUs, which is between 23% (memory-bound) and 29% (compute-bound)¹. This of course applies to Cases (a) and (c), too.

Therefore the previous work’s MPMG performance on G2 can be estimated from the performance of Case (c) and the performance ratio between (a) and (b), called SOSp, within

¹The differences in compute power (GLOPS) and memory bandwidth between M2090 and C2070 are 29% and 23% respectively

reasonable error margin, which we expect to be no more than 12% ($= (29\%-23\%)*2$).

CHAPTER V

Future Work

In this section we discuss our future work and some optimization possibilities. We observe that filters like splitters and joiners do not manipulate their input data. Their objective is either data distribution (in case of splitters) or consolidation (in case of joiners). When these kinds of filters are mapped to GPU with our method, their duties are simple since they only re-arrange GPU shared memory. While they do not have any effect on the data, their run-time contribution is significant. Elimination of the unnecessary splitters is rather easy, we only have to re-adjust the index of input data of the follow-up filter as depicted in Figure 5.1. However, if the joiner is excluded, the next filter that follows that joiner will have to handle the fragmentation problem as shown in Figure 5.2. This problem leads to more complex access pattern of the input data. We modify our buffer allocation algorithm and code generation to adapt to this enhancement. The preliminary results are shown in Table 5.1, in which we report the run-time of the enhanced version versus the original version when mapping Bitonic and FFT to 1 GPU. Bitonic has a relatively high number of splitters and joiners while FFT only has one splitter and one joiner. Elimination of splitters and joiners boosts the SPSG by $1.56\times$ with FFT and $2.86\times$ with FFT.

Figure 5.1: Shared memory map and filter operation of split structure. (a) an example of split structure. (b) original version. (c) enhanced version.

Figure 5.2: Shared memory map and filter operation of join structure. (a) an example of join structure. (b) original version. (c) enhanced version.

Table 5.1: Runtime(ms) comparison of original and enhanced version.

	N	Original	Enhanced	Speedup
FFT	512	39.2	27.2	1.44
	256	15.71	9.46	1.66
	128	8.28	5.2	1.59
Bitonic	64	23.14	5.2	4.45
	32	6	1.2	5.01
	16	0.98	0.94	1.05

CHAPTER VI

Conclusion

In this paper we presented a new partitioning-based mapping scheme for stream graphs. Graph partitioning is essential when mapping large graphs or mapping to multi-GPU platforms. At the same time, finding a good partition of a stream graph is challenging, partly because of the uncertainty in the ensuing mapping step. To address this challenge we propose a novel partitioning heuristic, which actively uses our accurate GPU performance model to make the most performance-enhancing partitioning decisions. Though it is limited due to its greedy nature, it is somewhat alleviated by exploiting the structure of the stream graphs. For multi-GPU targets, communication can easily become a performance bottleneck, determining the application throughput. To address this challenge we propose an ILP-based optimization scheme that can explicitly model the effect of communication, and is scalable up to graphs of hundreds of filters. Our experimental results on a real hardware platform demonstrate that our technique can generate scalable performance for up to 4 GPUs, achieving 3.2 times speedup on average for large stream graphs compared to 1-GPU mapping, and can generate mappings that are far better than the state of the art for compute-bound applications.

References

- [1] Yoonseo Choi, Yuan Lin, Nathan Chong, Scott Mahlke, and Trevor Mudge. Stream compilation for real-time embedded multicore systems. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society. 6
- [2] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. *SIGARCH Comput. Archit. News*, 34(5):151–162, October 2006. 6
- [3] A. Hagiescu, Weng-Fai Wong, D.F. Bacon, and R. Rabbah. A computing origami: Folding streams in fpgas. In *Proc. DAC*, pages 282–287, 2009. 6
- [4] Andrei Hagiescu, Huynh Phung Huynh, Weng-Fai Wong, and Rick Siow Mong Goh. Automated architecture-aware mapping of streaming applications onto gpus. In *Proc. IPDPS 2011*, pages 467–478. IEEE, 2011. 1, 5
- [5] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. Sponge: Portable stream programming on graphics engines. *SIGARCH Comput. Archit. News*, 39(1):381–392, March 2011. 1, 5
- [6] http://www.gurobi.com/products/gurobi_optimizer. 19

REFERENCES

- [7] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. Scalable framework for mapping streaming applications onto multi-gpu systems. *SIGPLAN Not.*, 47(8):1–10, February 2012. [vi](#), [2](#), [5](#), [8](#), [9](#), [16](#), [19](#), [21](#), [22](#), [23](#)
- [8] Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. *SIGPLAN Not.*, 43(6):114–124, June 2008. [6](#)
- [9] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, 2002. [3](#), [19](#)
- [10] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. pages 200–209, 2009. [4](#), [5](#), [6](#), [23](#)
- [11] Yongpeng Zhang and Frank Mueller. Gstream: A general-purpose data streaming framework on gpu clusters. *2013 42nd International Conference on Parallel Processing*, 0:245–254, 2011. [5](#)