Scaling Large-Data Computations on Multi-GPU Accelerators

Amit Sabne
Purdue University
asabne@purdue.edu

Putt Sakdhnagool
Purdue University
psakdhna@purdue.edu

Rudolf Eigenmann
Purdue University
eigenman@purdue.edu

ABSTRACT

Modern supercomputers rely on accelerators to speed up highly parallel workloads. Intricate programming models, limited device memory sizes and overheads of data transfers between CPU and accelerator memories are among the open challenges that restrict the widespread use of accelerators. First, this paper proposes a mechanism and an implementation to automatically pipeline the CPU-GPU memory channel so as to overlap the GPU computation with the memory copies, alleviating the data transfer overhead. Second, in doing so, the paper presents a technique called *Computation Splitting*, *COSP*, that caters to arbitrary device memory sizes and automatically manages to run out-of-card OpenMP-like applications on GPUs. Third, a novel adaptive runtime tuning mechanism is proposed to automatically select the pipeline stage size so as to gain the best possible performance. The mechanism adapts to the underlying hardware in the starting phase of a program and chooses the pipeline stage size. The techniques are implemented in a system that is able to translate an input OpenMP program to multiple GPUs attached to the same host CPU. Experimentation on a set of nine benchmarks shows that, on average, the pipelining scheme improves the performance by 1.49x, while limiting the runtime tuning overhead to 3% of the execution time.

Categories and Subject Descriptors: D.3.4 [*Programming Languages*] : Processors – Compilers

Keywords: GPU, Large-Data, OpenMP, Tuning, Out-of-card Computations, Pipelining

1. INTRODUCTION

Accelerators have become the forerunners of high-performance computing. Many supercomputers now use GPU devices as accelerators. Among many open issues are those related to programming models and program optimization. The present paper addresses these issues.

First, even though accelerators can be used as independent computational devices, they commonly serve as co-processors. Eligible computation is offloaded from the CPU - either explicitly by the programmer, or implicitly by the system software/hardware. Offloading involves data transfer from the CPU to the accelerator, which causes significant overhead – up to 95% of the program execution time, in our experiments. The primary contribution of this paper is an automatic pipelining generation technique that reduces this overhead. To do so, the pipelining technique overlaps data transfer with computation. It creates opportunities for such overlap by transforming the computation into multiple chunks and transferring the data for chunk '$(i+1)$' while executing chunk 'i'. Our technique contrasts with those that reduce data transfer overhead by eliminating redundant memory transfers [1, 2] and by advancing or delaying the data copy operations [3].

Second, the pipelining technique builds on an enabling technique that deals with another important issue in accelerators: The accelerator's memory space is limited; computation that fits in the CPU's memory may exceed the accelerator's capacity. The simple-most form of the technique splits computation into blocks that fit in memory. The same technique can be a key enabler for optimizations that tend to increase memory demand. Examples of such optimizations are data privatization, prefetch, and our pipelining technique. The difficulty in designing the technique is to model the increase in memory demand and determine the maximum chunk size that fits in the device memory. While most common accelerator benchmarks use data sizes that fit in memory, researchers have recognized the issue of limited memory sizes as well. For example, Liang [4] et al. discuss an example of an out-of-card FFT computation; a set of map-reduce based systems [5, 6] need to handle large data sizes that can exceed the GPU memory size. With the evolution of big-data systems, we expect the computing focus to move to large datasets. Our technique is the first to automatically tailor computation to the available memory space while considering optimizations that increase memory demand. The ability of splitting the computation also provides an opportunity to perform multi-device mapping of the program. Multiple GPUs attached to a single computation node are becoming an architectural reality. Several techniques have been proposed [7, 8] to port custom applications to multi-GPUs. Our framework automates this process.

The third issue addressed in this paper is the programmability of accelerators. An important issue is the design of a suitable high-level programming model, with one of the key

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

questions being what architectural details need to be exposed to the programmer. Among many proposed models [9, 10, 2, 11, 12, 13], recently, the idea of using OpenMP extended with directives for accelerators, has gotten traction. We will integrate and evaluate our techniques in one of the most advanced compilers that has been pursuing this idea, OpenMPC [2]. In doing so, we introduce a novel component that addresses a fundamental problem in high-level programming environments for accelerators: The architectural complexity of the CPU-Accelerator systems exacerbates the difficulty of advanced compilers in making optimization decisions that need runtime information. These architectural intricacies are detrimental to the portability of the code. An optimally tuned program on one platform may perform poorly on another. The most advanced compilers make use of *offline tuning* techniques to obtain the optimal choices of system-specific parameters. By contrast, we describe an adaptive runtime tuning mechanism that learns about the architectural details in the initial phase of the program in a short time and determines the most suitable pipeline stage size to attain best performance.

In this paper, we make the following specific contributions:

- We design and implement an automatic pipelining technique that reduces CPU-accelerator data transfer overhead by overlapping the data transfers with computation. We will demonstrate the efficacy of this technique by displaying the performance benefits achieved on a set of benchmarks, including kernels and applications. On average, pipelining achieves a speed-up of 1.49x over the baseline OpenMPC codes.
- We design and implement an automatic computation splitting technique, *COSP*, that fits large computations into the accelerator's device memory. In doing so, it considers program transformations that increase memory demand, including our pipelining technique. We will show that large, out-of-card data sizes that can not be otherwise handled by OpenMPC/CUDA can be successfully run.
- We describe a low-overhead (less than 3% execution time) adaptive runtime tuning method that chooses a good split size for a problem and the underlying hardware so as to approach the best pipelining performance.
- We describe a system containing our novel techniques that automatically translates an input OpenMP code into a multi-device CUDA code that can employ multiple GPUs attached to the same host node. We evaluate the performance of benchmarks executed on multi-GPU systems.

The remainder of the paper is organized as follows : Section 2 provides background information about GPGPU programming and OpenMPC. Section 3 analyzes the benefits of *COSP* and describes implementation details. Section 4 explains how pipelining and multi-GPU code generation are enabled by *COSP* and provides compiler design details for both. Section 5 describes the proposed adaptive runtime tuning system. Finally, Section 6 evaluates our system on a set of benchmarks from the StreamIt benchmark suite, CUDA SDK and Rodinia benchmarks.

2. PRELIMINARIES

This section describes the GPU system architecture and the CUDA programming model. We also describe the OpenMPC compiler system that translates an input OpenMP program into CUDA code.

2.1 GPUs and CUDA

Typically, one or more GPUs are connected to a Host CPU via a PCIe bus. The CPU can offload computation kernels onto the GPU(s). Since the CPU and the GPU have different memories, input data must be copied from the CPU to the GPU before the start of a kernel. This operation is called *copy-in*. Similarly, copying the outputs of a kernel from the GPU to the CPU is termed *copy-out*. CPU-GPU data transfers are initiated and governed by the CPU.

GPUs are SIMD units with a large number of processors. NVIDIA GPUs have termed these processors *Streaming Multiprocessors* or *SMs*, each being an SIMD processor. CUDA [9] is a multi-threaded, SIMD programming model. The threads on the GPU device are divided into *ThreadBlocks*. Each *ThreadBlock* consists of a set of threads, each executing the same code. When the CPU launches a kernel, it prescribes the number of threads in a *ThreadBlock* along with the number of *ThreadBlocks* to launch. The set of *ThreadBlocks* launched by a kernel is called a *grid*.

CUDA has a complicated memory hierarchy. Each thread has its own local memory and a set of registers. Local memory contains a stack which is primarily used to spill the registers. A *ThreadBlock* has its on-chip local storage in the form of *shared memory*. The off-chip global memory is accessible to all threads in the grid. Further, on-chip *constant* and *texture* memories can act as read-only buffers.

Various factors impact the performance of a GPU kernel: coalesced memory accesses, register and shared memory usage, number of threads in a *ThreadBlock*, shared memory bank conflicts etc., making it difficult to predict the GPU performance [14, 15, 16]. To obtain good performance results, tuning is essential for GPU programs.

2.2 OpenMPC

OpenMPC [2] is a programming framework that synthesizes CUDA programs from OpenMP codes. The framework includes an extended OpenMP programming interface, a source-to-source translator [17], and an automatic compiler-assisted tuning system. The programming interface extends OpenMP with a new set of directives and environment variables for controlling CUDA specific parameters and optimizations. OpenMPC applies various code transformations and CUDA extensions to the input OpenMP code.

We chose OpenMPC as the underlying compiler for our system implementation because of the following reasons : (a) OpenMP-like programming models are becoming popular, especially with the advent of OpenACC [13]. The OpenMP standard itself is in the process of being extended to support accelerator devices [18]. OpenMPC is a research framework pursuing the same idea. (b) OpenMPC automatically performs many safe and beneficial code transformations so as to coalesce memory accesses and map data to different kinds of memories available on the GPU. OpenMPC also implements sophisticated CPU-GPU live variable analysis to remove or hoist the redundant memory transfers. (c) Considering the complexity of the CUDA programming model, its not always possible for a compiler to find the performance optimal CUDA parameters. OpenMPC provides OpenMP extensions for the programmer that can be

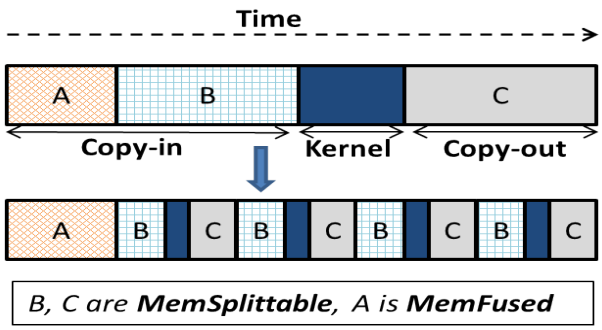

Figure 1: *COSP* - *MemSplittable* data can be split since only a part of them is required per chunk of computation, *MemFused* is the part of the data accessed by every chunk.

used to set the CUDA related parameters. (d) The OpenMPC translator is realized on top of the Cetus [19] compiler, which provides an efficient implementation infrastructure.

3. COSP - AN ENABLER TECHNIQUE

COSP, or *Computation Splitting*, divides a given problem into smaller, homogeneous subproblems. It acts as an enabler technique to other optimizations. In this section, we describe the nature of *COSP* and the optimizations it enables. While doing so, we establish an analytical upper bound on the size of a problem that can be run on an accelerator with limited memory. We provide a lower bound on the number of splits that the problem must undergo in order to fit in the device memory. We also describe the implementation of *COSP*.

3.1 Catering to Arbitrary Device Memory Sizes

COSP reduces the runtime device memory requirement of a problem; every subproblem requires less memory than the overall computation. We now analyze the factors that impact the device memory requirement. The total device memory requirement of a kernel depends upon the following:

- *COSP* converts a large problem into smaller subproblems. An example of such operation is shown in Fig. 1. In this example, data A, B, and C are a part of OpenMP *'shared'* data objects i.e. all threads work on a single copy of the data. In the original computation, data A and B are copied-in. The kernel uses these elements to generate data C, which is copied out of the device memory. The kernel, even after splitting, requires all data A for generating even a part of data C; we call A *MemFused* type of data. On the other hand, data B and C can be split, and each subproblem only requires a part of these elements. We call B and C *MemSplittable* data. If the splitting is perfect, amongst $numSplits$ subproblems, the original runtime device memory space for the shared data goes down from $(MemSplittable + MemFused)$ to $(MemSplittable/numSplits + MemFused)$. However, a subproblem may also use the data of another (mostly neighboring) subproblem. We model this extra data requirement overhead by *SplitOverlap*.
- Every OpenMP *Private* data element in the input OpenMP program has to be allocated per thread in the computation. If the size of the private element is small (e.g. scalars), the element is generally stored in the device register. However, if the size of the private element is large (e.g. for arrays), the element needs to be placed in the local memory of the thread. Therefore, if P is the size of all such elements together and $NumThreads$ is the number of total threads, the device memory requirement is $P \times NumThreads$. Further, if the problem is split amongst $numSplits$ subproblems, the device memory requirement for the private data would be $P \times NumThreads/numSplits$.
- Memory prefetching is an important technique in accelerator programming. Device memory prefetching advances the copy-in operation for the ready data so as to overlap the copying time with the CPU computation. Similarly, for the data that is not immediately required by the CPU, the copy-out can be delayed while overlapping the copy-out time with the CPU computation. Both these optimizations require device memory to hold the buffered data. We denote this size as $PrefBuffer$.

To ensure that the device memory, $DevMem$, can fit all of the above components, the following constraint must be met:

$$DevMem \geq \left(P \times \frac{NumThreads}{numSplits} + PrefBuffer + MemFused + \frac{MemSplittable}{numSplits} + SplitOverlap\right)$$

$$\Rightarrow numSplits \geq \left\lceil \frac{(MemSplittable + P \times NumThreads)}{(DevMem - PrefBuffer - MemFused - SplitOverlap)} \right\rceil \quad (1)$$

Equation 1 provides a lower bound on the number of splits required on the input problem to make it fit in the device memory. Equation 1 also indicates that the benefits of *COSP* are multi-faceted. *COSP* can enable an accelerator program to run successfully even when the OpenMP *private* elements push the device memory requirements beyond available. *COSP* can also partition the computation into appropriate subproblems so as to enable prefetching in the device memory.

3.2 COSP Through a Code Example

Section 3.1 provided a discussion on the lower bound of the number of splits required by the computation so as to make it fit in the device memory. We now describe a mechanism to achieve *COSP* through a compiler transformation. If all the parameters in Eq.1 are known, the compiler can choose the number of splits required to make the computation fit in the device memory. As we would show later in Section 4, *COSP* creates an opportunity for pipelining the subproblems. The number of subproblems required to obtain efficient pipelining can be less than the upper bound provided by Eq.1. Our compiler therefore abstracts out the number of splits as a variable.

OpenMP programs usually encapsulate parallelism using large *for* loops. We introduce the *COSP* mechanism through an example of *Scalar Product* code, as shown in Listing 1. This program generates scalar products for a set of vectors. The split version of the code generated by our compiler is shown in Listing 2. The input *parallel for* loop is split into many small loops. The upper bound for the inner *for* loop

is set to be *SplitSize*. SplitSize is left as a parameter that can be set by the user or the automated tuning system. The inner loop body represents a subproblem of the initial large problem. We hereafter refer to the inner loop as a ***Split*** and the outer loop as ***Split Loop***.

Listing 1: Input Scalar Product OpenMP Code

```
#pragma omp parallel for shared(D, E, F)
private(vec, pos, sum)
for(vec = 0; vec < NUM_VECTORS; vec++) {
  sum = 0;
  for(pos = 0; pos < NUM_ELEMENTS; pos++) {
    sum += D[NUM_ELEMENTS * vec + pos ] *
    E[NUM_ELEMENTS * vec + pos];
  }
  F[vec] = (float)sum;
}
```

Listing 2: Split Parallel Region for Scalar Product

```
for (split=0; split < NUM_VECTORS/SplitSize; split = split+1) {
  #pragma omp parallel for shared(D, E, F)
  private(vec, pos, sum) shared(split, SplitSize)
  for (vec = 0; vec < SplitSize; vec++ ) {
    sum=0;
    for (pos = 0; pos < NUM_ELEMENTS; pos++ ) {
      sum+=(D[(pos + (NUM_ELEMENTS *
      (vec + split*SplitSize)))]*
      E[(pos + (NUM_ELEMENTS * (vec + split*SplitSize))]);
    }
    F[(vec + split*SplitSize)] = (float)sum;
  }
}
```

For the *Scalar Product* program, the total number of splits created is equal to $NUM\_VECTORS/SplitSize$. This number needs to be larger than the *numSplits* calculated by Eq. 1 in Section 3.1 so as to meet the device memory size constraint. The number of splits in the problem is controlled by the SplitSize. Further, SplitSize governs the data size required by the kernel as well as the number of GPU threads synthesized by the OpenMPC system. Note that the *COSP* version in Listing 2 does not explicitly formulate the data partitions per Split. We defer the data partitioning analysis till the next section.

COSP can lead to extra data copy overheads in cases where the computation splitting is not perfect. As an example, consider stencil programs where computing each output requires an array element along with its neighbors. In such cases, *COSP* would require to account for the storage of boundary elements of each split, increasing the amount of overall data copied from the CPU to the GPU.

Since the subproblems provided by *COSP* are devoid of data dependences, they can be run in parallel. *COSP* can therefore generate opportunities for pipelining the slow host-device memory channel in an accelerator-based system. It also offers an easy- to-distribute program structure to the compiler system, which can then map subproblems to different devices.

4. PIPELINING

Pipelining, in general, is a throughput-enhancing technique that overlaps the execution phases of one computation with another so as to make the best use of the available computational resources. We introduce an example of pipelining in Fig. 2. After performing *COSP*, one Split's kernel execution is overlapped with the next Split's memory copies; i.e. while the first subproblem kernel is working on the already copied-in data B, the copy-in for the next subproblem kernel's data B is taking place. It is important to note that to achieve successful pipelining, there should be two buffers present for each *MemSplittable* data so as to obtain the necessary prefetching.

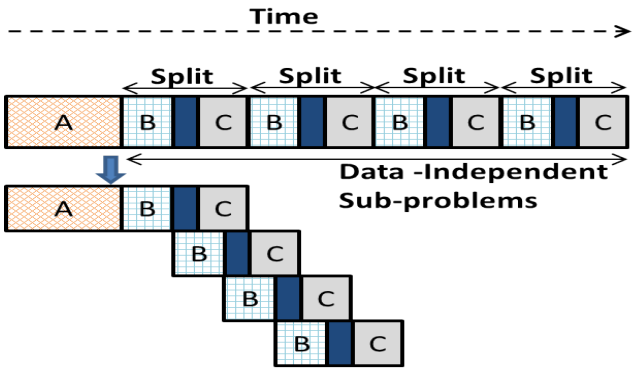

Figure 2: Pipelining Opportunity Generated by *COSP* : Individual Splits are independent; they can be pipelined

In the case at hand, the three resources that we propose to pipeline include the CPU-GPU channel, GPU-CPU channel (if different than the first) and the GPU computational units. This section focuses on the pipelining code generation of the compute-split code. It also explains the strategy to perform multi-device code mapping and implementation mechanisms adopted by our compiler.

4.1 Achievable Speedup from Pipelining

Maximum attainable speedup from pipelining is restricted by the number of pipelining stages. In the proposed scheme, there are three pipeline stages : (i) Memory channel between the CPU and the GPU (ii) Memory channel between the GPU and the CPU (iii) GPU Computation cores. Hence, the pipelining speedup can be at most three. Most new GPUs support overlaps between the computation and transfers. Some advanced GPUs, such as Tesla M2090, also support overlaps in the memory copy operations in different directions, since they have different channels for copying in each direction. For GPUs without dedicated copy engines, the speedup would be restricted by two.

$$Speedup = \frac{t_{compute} + t_{MemFused} + t_{co} + t_{ci}}{t_{MemFused} + max(t_{compute}, t_{co}, t_{ci})} \quad (2)$$

where
$t_{compute}$ = Time spent in kernel computation
$t_{MemFused}$ = Time spent in transferring MemFused data
t_{ci} = Time spent for copy-in of MemSplittable data
t_{co} = Time spent for copy-out of MemSplittable data

Equation 2 provides an upper bound on the pipelining speedup, assuming the *COSP* overhead is zero. It also assumes the presence of different memory copy channels for copy-in and copy-out operations.

All outputs of a parallel program fall under the *MemSplittable* category unless they are reductions. Although *COSP* can always 'split' the outputs of a program, it may not be always able to do so for its inputs, if the inputs fall under the *MemFused* category. The worst case scenario would be a program wherein computation of a single output element requires input data of type *MemFused* that has a size larger than the device memory. In such a case, algorithmic change in the program structure is the only alternative.

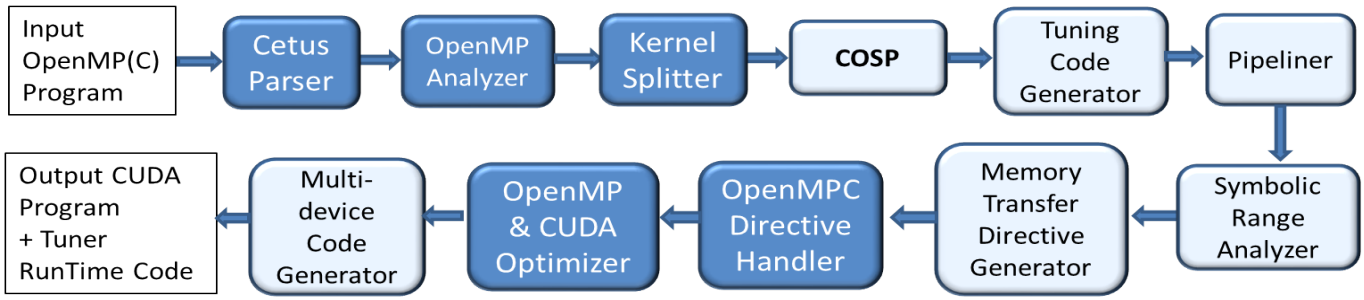

Figure 3: Overall System Flow : Darker boxes indicate the base OpenMPC passes.

4.2 Compiler Organization

Figure 3 shows the structure of our compiler. It builds on OpenMPC, representing the program using Cetus IR [19]. After parsing the source code, OpenMPC system-internal decision making is performed to identify the kernels to be offloaded to the GPU. *COSP* is performed next on the kernel regions recognized by the OpenMP analyzer. For each eligible kernel region, tuning code is generated. The pipelining pass is applied next. Pipelining is optional in the sense that a user could simply generate only the compute-split code. Communication generation for each pipeline chunk is handled by the advanced symbolic range analysis [20] stage, which encodes its results as OpenMPC memory transfer pragmas. Once the OpenMPC directive handler and CUDA optimizer have finished their work on the IR, the multi-device code generation pass maps the pipelines to their respective devices. For the correct functioning of the multi-device code generation pass, the pipelining pass is made multi-device aware, in the sense that it keeps a map of the pipelines to devices, which is utilized later on by the multi-device code generation pass.

4.3 Generating Pipelined Code

At the heart of realizing the pipelining is the ability of the CPU to execute memory copy and kernel operations on the device asynchronously. The underlying CUDA model provides a mechanism called *'CUDA streams'* that realizes this asynchronous operation. A *'cudaStream'* [9] represents an instruction stream of computation on the host CPU that queues the launches of device commands i.e. kernel launches and memory transfers. All operations on a given *cudaStream* are launched sequentially, however, different *cudaStreams* can run independently of each other. Our pipelining implementation describes the strategy using *cudaStreams*.

The pipelining stage unrolls the Split Loop in Listing 2 twice. Listing 3 displays the intermediate code generated by our compiler. The unrolling factor of two corresponds to the two data buffers required; one for the currently executing Split and another for prefetching the inputs of the next Split. Each unrolled Split is transformed into a separate kernel and the required memory buffers are allocated for each kernel by the base OpenMPC system. Unrolling the Split Loop eases the software pipelining implementation.

Bounds for the data required by each Split need to be precisely calculated to avoid transferring more than necessary data (lines 8-13 in Algo. 1). We develop an aggregate data sections analysis using the advanced symbolic range analysis techniques offered in Cetus. This algorithm determines the starting locations and data lengths per Split required for all aggregate data elements.

Listing 3: Intermediate Representation for Pipelining the Compute Split Parallel Region

```
for (split=0; split<NUM_VECTORS/SplitSize; split=split+2) {
  //Determine starting points and ranges required per Split
  E_c2gstart_stream_0=((split*NUM_ELEMENTS)*SplitSize);
  E_c2grange_stream_0=(NUM_ELEMENTS*SplitSize);
  F_g2cstart_stream_0=(split*SplitSize);
  F_g2crange_stream_0=SplitSize;
  D_c2gstart_stream_0=((split*NUM_ELEMENTS)*SplitSize);
  D_c2grange_stream_0=(NUM_ELEMENTS*SplitSize);
  E_c2gstart_stream_1=((NUM_ELEMENTS*SplitSize)+
  ((split*NUM_ELEMENTS)*SplitSize));
  E_c2grange_stream_1=(NUM_ELEMENTS*SplitSize);
  F_g2cstart_stream_1=(SplitSize+
  (split*SplitSize));
  F_g2crange_stream_1=SplitSize;
  D_c2gstart_stream_1=((NUM_ELEMENTS*SplitSize)+
  ((split*NUM_ELEMENTS)*SplitSize));
  D_c2grange_stream_1=(NUM_ELEMENTS*SplitSize);

  //Code to be launched by cudaStream 0
  #pragma omp parallel for  private(pos, sum, vec) shared \
  (NUM_ELEMENTS, NUM_VECTORS, split, SplitSize, D, E, F)
  #pragma cuda gpurun \
  g2cmemtr(F[F_g2cstart_stream_0: F_g2crange_stream_0]) \
  c2gmemtr(D[D_c2gstart_stream_0: D_c2grange_stream_0]) \
  c2gmemtr(E[E_c2gstart_stream_0: E_c2grange_stream_0])
  #pragma cuda gpurun noc2gmemtr(F) \
  nog2cmemtr(NUM_ELEMENTS, NUM_VECTORS, D, E)
  for (vec=0; vec<SplitSize; vec ++ ) {
    sum=0;
    for (pos=0; pos<NUM_ELEMENTS; pos ++ ) {
      sum+=(D[(pos+(NUM_ELEMENTS*
      (vec + split*SplitSize)))]*
      E[(pos+(NUM_ELEMENTS*(vec + split*SplitSize)))]);
    }
    F[(vec + split*SplitSize)]=((float)sum);
  }

  //Code to be launched by cudaStream 1
  #pragma omp parallel for  private(pos, sum, vec) shared \
  (NUM_ELEMENTS, NUM_VECTORS, split, SplitSize, D, E, F)
  #pragma cuda gpurun \
  g2cmemtr(F[F_g2cstart_stream_1: F_g2crange_stream_1]) \
  c2gmemtr(D[D_c2gstart_stream_1: D_c2grange_stream_1]) \
  c2gmemtr(E[E_c2gstart_stream_1: E_c2grange_stream_1])
  #pragma cuda gpurun noc2gmemtr(F) \
  nog2cmemtr(NUM_ELEMENTS, NUM_VECTORS, D, E)
  for (vec=0; vec<SplitSize; vec ++ ) {
    sum=0;
    for (pos=0; pos<NUM_ELEMENTS; pos ++ ) {
      sum+=(D[(pos+(NUM_ELEMENTS*
      (vec + (split+1)*SplitSize)))]*
      E[(pos+(NUM_ELEMENTS*(vec + (split+1)*SplitSize)))]);
    }
    F[(vec + (split+1)*SplitSize)]=((float)sum);
  }
}
```

It also categorizes the data into *MemSplittable* and *MemFused* categories. The algorithm determines the range of access for each usage of the given variable. It then performs a union operation on the individual ranges to get the

comprehensive access range as well as the starting address of the data. Both these expressions are symbolic in nature and are parameterized by *SplitSize* and *Split* (Lines 3-17 in Listing 3).

The starting address and length for data copy are specified to the OpenMPC system using *c2gmemtr* and *g2cmemtr pragma*s that govern the generation of CPU-to-GPU and GPU-to-CPU communication, respectively. (Lines 23-25 and 42-44 in Listing 3). For *MemFused* data elements, for every Split, entire aggregate datatype transfer is required. In such cases, transfers for these data elements are hoisted outside the Split Loop.

Algorithm 1 Pipelined Code Generation from Compute Split Program

Input: Compute Split Loop '*Region*'
Output: Pipelined *CUDA* code

```
1: Unroll Region by a factor of 2;
2: Create cudaStreams stream0, stream1;
3: StreamMap = new Map(cudaStream, Split);
4: StreamMap.insert(stream0, first Split in Region);
5: StreamMap.insert(stream1, second Split in Region);
6: For each split ∈ Region do
   //Static code contains only two Splits
7:    For each sharedVar ∈ split do
   //sharedVar is OpenMP shared type
8:       range = rangeAnalysis(sharedVar , split);
9:       if (range contains SplitSize) then
10:         start = getStartingPoint(range);
11:         insertCopyPragmas(sharedVar, start,
   range);
12:      else
   //this is a MemFused Variable, hoist memory
   //transfers out of the Split Loop
13:         insertCopyPragmas(sharedVar, Region);
14:      end if
15:   end for
16: end for
17: cudaCode = translate(Region, StreamMap);
   //Format CUDA code to achieve desired
   //queueing using streams
18: reOrganize(cudaCode, stream0, stream1);
```

Since GPUs have at most one data copy engine in the CPU-to-GPU or GPU-to-CPU direction, it is necessary to schedule the operations on this channel wisely in order to avoid bottlenecks on the data copy engines. Correct scheduling is necessary to assure maximum overlapping benefits as well. The first Split in the unrolled Split Loop is assigned to *cudaStream 0*. That is, the kernel launches and memory transfers for this Split are handled by *cudaStream 0*. Similarly, the second Split is attached to *cudaStream 1*. In this manner, unrolling of the Split loop helps in (a) Creating private buffers on the device per *cudaStream* and (b) Providing an easy kernel- *cudaStream* mapping. Memory copy requests in a given direction from both *cudaStreams* get serialized. Hence, the corresponding queueing strategy performs copy-in from *cudaStream 0* for Split 1, then issues the kernel from *cudaStream 0* for Split 1 and simultaneously performs copy-in via *cudaStream 1* for Split 2. Next, the system launches the kernel from *cudaStream 1* and subsequently issues copy-out from *cudaStream 0* and *cudaStream 1* respectively. Listing 3 shows that alternate Splits go on different *cudaStreams*.

Our system maintains a mapping between a Split and its corresponding *cudaStream*. After the CUDA code is generated by OpenMPC, the compiler reorganizes the memory transfer and kernel launches so as to realize the queuing strategy. We portray the complete pipelining code generation algorithm in Algorithm 1.

4.4 Multi-GPU Code Generation

As seen in the previous section, to implement pipelining, two *cudaStreams* are required for a single GPU. For multi-device code generation, our system extends this strategy and assigns two *cudaStreams* per device, the overall number of *cudaStreams* used being twice the number of devices. In other words, the Split Loop is unrolled twice per device.

The next important stage in multi-device code generation is to perform work partitioning amongst devices. We use block-cyclic partitioning wherein two contiguous Splits are attached to the same device, since these Splits are more likely to access data elements in the vicinity of each other; this method improves spatial locality while performing CPU-GPU transfers.

Further, data elements that need to be present completely inside the device memory even for a single Split computation (*MemFused* elements) need to be made 'private' per device, and the memory copies for such elements need to be hoisted out of the Split Loop. An example with the overall multi-device code generation strategy for two devices is shown in Fig. 4. Note that the dependences are caused due to the queueing on the memory copy channels.

5. ADAPTIVE RUNTIME TUNING SYSTEM

SplitSize is the number of iterations of the parallel loop in a given Split. SplitSize can therefore be thought of as the size of the pipeline stage. Each iteration of the Split is mapped to a GPU thread by OpenMPC. Hence, the choice of SplitSize determines the number of $ThreadBlocks$ issued per Split, governing both the time required for computation and CPU-GPU communication. In this section, we describe how the choice of SplitSize impacts the pipelining performance. We then propose a heuristic tuning algorithm that selects the most suitable SplitSize.

5.1 Performance Variation with SplitSize

Equation 1 gave the maximum SplitSize that fits in the device memory. The SplitSize that yields the best performance results is usually less than this upper bound. Fig. 5 shows such execution time variation with different SplitSizes for two benchmarks. Experiments were run on Tesla M2090, which has separate memory copy engines in CPU-to-GPU and GPU-to-CPU directions. *Vector Add* is a memory copy-intensive application, whereas *Filterbank* is a compute-intensive one. *Vector Add* results were generated for problem size (iteration space) of 2^{27}; for *Filterbank*, the problem size was 2^{24}. Fig. 5 shows that the performance becomes better with increasing SplitSize for *Vector Add*, while better results can be achieved at lower SplitSize values for *Filterbank*. The choice of SplitSize is therefore important to achieve good performance, but selecting the correct SplitSize is not straightforward for the programmer. To build an intuition for choosing the SplitSize, we begin by analyzing the performance results on both memory copy-intensive and compute-intensive programs. The former spend most

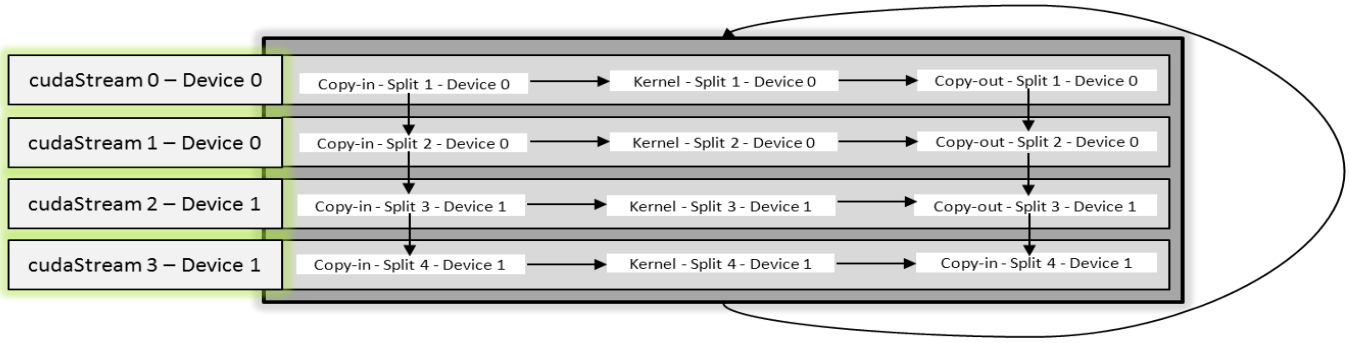

Figure 4: System Strategy to generate and run programs on Multiple Devices - In this case, the number is 2. Straight arrows depict dependences, the curved arrow represents Split loop

of their execution time on CPU-GPU transfers. The latter spend most of their execution time running the GPU kernels.

Fig 6 shows examples of the different behaviors of compute-intensive and memory copy-intensive programs. *Monte Carlo* is compute-intensive, while *Black Scholes* is memory copy-intensive. SplitSizes are chosen such that the number of *ThreadBlocks* launched by each SplitSize is a multiple of the number of SMs, *SMCount*, of the GPU. Note that the memory copy times show linear increase at smaller SplitSizes as compared to the kernel execution times. Since the pipelining benefits are higher when the pipelining stages are of equal size, relative increase in kernel execution time as compared to the memory copy time leads to better performance. Once the kernel execution starts to grow linearly with SplitSize, there is no more opportunity for higher benefits. This explains the performance variation with SplitSizes for *Vector Add* and *Black Scholes* benchmarks. Both these benchmarks have the highly memory copy-intensive *Type 1* overlapping pattern as shown in Fig. 7.

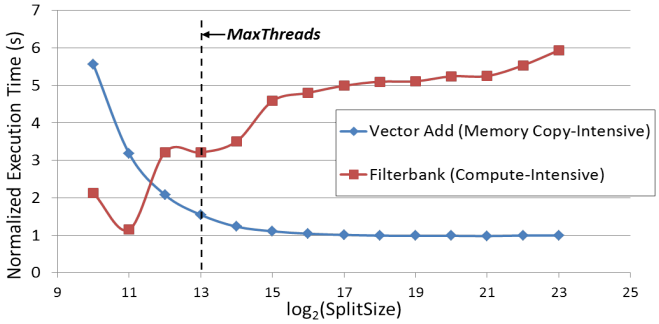

Figure 5: Performance Variation with SplitSize : Performance is higher for smaller SplitSizes for compute-intensive benchmarks, while for memory copy-intensive ones, performance is higher for larger SplitSizes. *MaxThreads* is the maximum number of threads that can co-exist on the GPU.

Compute-intensive programs like *Filterbank, Monte Carlo,* show the second type of overlap displayed in Fig. 7. In both these codes, the kernel execution time is larger than the sum of copy-in and copy-out times. A quick look at Fig. 7 shows that when the kernels are executing, the memory channel(s) are unused. One way to increase the kernel and memory copy overlap would be to run more *cudaStreams* so that the kernel time between the overlap of t_K^1 and t_K^2 (Fig. 7) can be further overlapped by memory transfers. Another alternative is to reduce the SplitSize and allow concurrent kernel execution, supported by the newer GPUs. Kernel-kernel

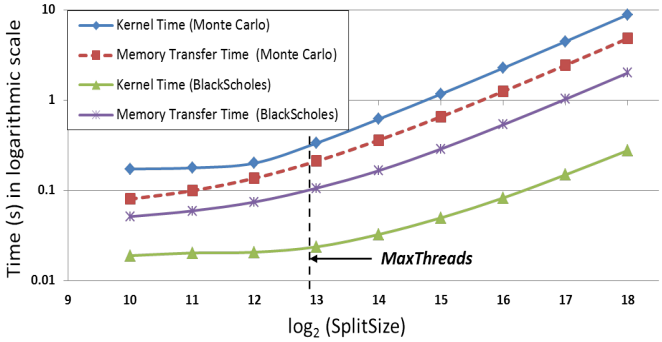

Figure 6: Kernel execution and Memory copy times per Split for different SplitSizes. Monte Carlo is compute-intensive; Black Scholes is memory copy-intensive. Experiments were run on Tesla M2090.

overlaps can easily occur in compute-intensive benchmarks. Due to the limited knowledge in GPU *ThreadBlock* scheduling mechanisms in the presence of multiple kernels, kernel-kernel overlap performance is difficult to predict or model and the performance tuning would need explicit runs with different SplitSizes. However, increasing the SplitSize to be larger than the maximum number of threads that can coexist on a GPU ($MaxThreads$) would only lead to queueing up of *ThreadBlocks* in the launching process. Hence, the SplitSize search space of interest in case of compute-intensive benchmarks is fairly small i.e. $SplitSize \leq MaxThreads$.

5.2 Adaptive Runtime Tuning Algorithm

With the aforementioned observations, we have developed a heuristic adaptive runtime tuning algorithm (Algorithm 2) for finding the SplitSize that would yield the best performance. First, the algorithm determines if the kernel is highly memory copy-intensive *(Type 1)* or highly compute-intensive *(Type 2)* or neither *(Type 3)* by running a pilot run with SplitSize equal to the $MaxThreads$.

Note that running just one Split (RUN_ONE_SPLIT, line 20 of Algo. 2) of the Split Loop is sufficient for the algorithm to determine the type of the kernel. The algorithm then generates a unique set of SplitSizes, named *CSet* or *configuration set*, depending upon the type of the program. Each SplitSize is chosen such that the number of *ThreadBlocks* contained in it is a multiple of $SMCount$, so as to evenly distribute the *ThreadBlocks* on SMs. The *CSet* generation is handled by the GENERATE_CONFIGS function (line 17 of Algo. 2). The largest SplitSize generated by the GENERATE_CONFIGS function is bounded by the device memory

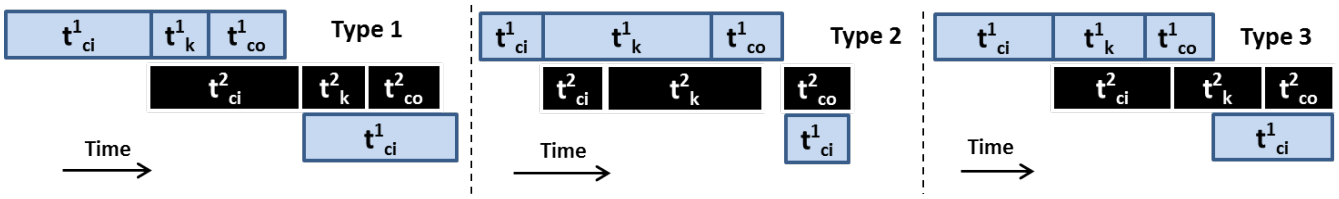

Figure 7: Possible Overlap Types for GPUs with Distinct Copy-in and Copy-out Engines : t_{ci} is the copy-in time, t_{co} is the copy-out time and t_k is the kernel execution time. Number in the superscript represents the *cudaStream*. Type 1 is the highly memory copy-intensive type with $t_{ci} > t_{co} + t_k$. Note that t_{ci} and t_{co} are interchangeable. Type 2 is highly compute-intensive, with $t_k > t_{ci} + t_{co}$. Type 3 is neither ($t_{ci} > t_k > t_{co}$ & $t_k + t_{co} > t_{ci}$). Note that shown is one of the many cases of different program patterns that can fit in *Type 3*.

capacity. It is worthwhile to note that the algorithm is agnostic of the number of threads per *ThreadBlock*.

Algorithm 2 Heuristic SplitSize Tuning Algorithm

Input: T is the *Number of Threads per ThreadBlock*
Input: $SMCount$ is the *Number of SMs*
Input: B is the maximum *ThreadBlocks* per SM
Output: Best SplitSize configuration

1:
2: //t_{ci} is copy-in time, t_k is kernel execution
//time, t_{co} is copy-out time
//t_o is the overlapped time : t_o =
//$min[max((t_{ci}+t_k), t_{co}), max((t_{co}+t_k), t_{ci})]$
//timer is an array of structures storing
//quadruplet t_{ci}, t_{co}, t_k, t_o
//$Normalizer(i) = MaxThreads/CSet(i)$
3:
4: **function** TUNER
5: $MaxThreads \leftarrow T \times SMCount \times B$
6: $Type \leftarrow$ Nil
7: $(t_{ci}, t_k, t_{co}, t_o) \leftarrow RUN\_ONE\_SPLIT(MaxThreads)$;
8: **if** $(t_{ci} + t_{co} \leq t_k)$ **then**
9: $Type \leftarrow 2$;
10: **else if** $(t_{ci} + t_k \leq t_{co} \vee t_{co} + t_k \leq t_{ci})$ **then**
11: $Type \leftarrow 1$;
12: **else**
13: $Type \leftarrow 3$;
14: **end if**
15: //GENERATE_CONFIGS creates a sorted set of
//SplitSize configurations
16: $CSet \leftarrow GENERATE\_CONFIGS(Type)$;
17: $min \leftarrow 1$;
18: **for** $i = 1 \rightarrow length(CSet)$ **do**
19: $timer[i] = RUN\_ONE\_SPLIT(CSet[i])$;
20: **if** $(Type = 1)$ **then**
21: **if** $(i > 1)$ **then**
22: $diff \leftarrow ((timer[i].t_k \times Normalizer(i)) - (timer[i-1].t_k \times Normalizer(i-1))$;
23: **if** $(|diff| \approx 0)$ **then return** $CSet[i-1]$;
24: **end if**
25: **end if**
26: **else if** $(timer[min].t_o \times Normalizer(min) < timer[i].t_o \times Normalizer(i))$ **then**
27: $min = i$;
28: **end if**
29: **end for**
30: **if** $(Type = 1)$ **then** **return** $CSet[i-1]$;
31: **else** **return** $CSet[min]$;
32: **end if**
33: **end function**
34:
35: **function** RUN_ONE_SPLIT(SplitSize)
36: Run one *Split* from the *Split loop*
37: Return corresponding t_{co}, t_{ci}, t_k, t_o
38: **end function**
39:

*This algorithm assumes bidirectional copy channels.

For *Type 1* programs, generated *CSet* contains SplitSizes $\geq MaxThreads$ in an increasing order. As explained earlier, a linear increase in the kernel execution time indicates the highest performance for memory copy-intensive programs. The algorithm therefore runs one Split per SplitSize in the *CSet* until the kernel time starts to grow linearly. Linear growth is established by comparing the kernel execution time of the previous SplitSize in the *CSet* (Line 23, in Algo. 2). This results in the selection of smallest SplitSize that would generate high performance while maintaining low device memory requirements.

For *Type 2* programs, algorithm generates the *CSet* with SplitSizes that are $\leq MaxThreads$. For each SplitSize, the runtimes for a single Split are noted, and the SplitSize with the best normalized overlapped time (Line 2, Algo. 2), t_o, is selected as the candidate.

Type 3 programs are neither highly compute-intensive nor memory copy-intensive. In such cases, the algorithm generates all possible SplitSizes (bounded by the device memory size) and runs one Split for each of them. The SplitSize with the best normalized overlapped time, t_o, is selected. *Type 3* denotes the worst case for the algorithm, since it requires a traversal in a larger space. However, most of the programs that we tried fell into either *Type 1* or *Type 2*.

Since the algorithm needs to run only a single Split to learn about the characteristics of a given SplitSize, the runtime overhead incurred is low.

5.3 Compiler Support for Tuning

The adaptive runtime tuning algorithm is automated in the compiler. The compiler automatically generates a tuning function for each kernel region. The tuning function contains an outlined copy of the parallel region, which is used to realize RUN_ONE_SPLIT function. This outlining is performed at the OpenMP level. Since the tuning function involves execution of some extra Splits, the original data may get modified. Outlining prevents this potentially harmful behavior to the kernel data during the tuning execution, as it forces the data elements in the tuning function to be allocated separately. Timer calls are inserted in this outlined copy to gather t_k, t_{ci} and t_{co}. Certain parts of the algorithm are inserted in this function, while others, such as GENERATE_CONFIGS, are implemented in a run-time library. The compiler automatically inserts the necessary calls to the runtime library functions. The tuning function is invoked only during the first call to the kernel; subsequent kernel calls use the SplitSize value that was generated during the first run.

Notice that the tuning system may not represent the best SplitSize if the control flow of the parallel region is divergent. However, if the parallel region is invoked only once,

the first invoke runs the tuner which indeed finds the correct SplitSize.

6. EVALUATION

This section evaluates the performance of the presented computation splitting, pipelining and multi-device code generation regimes. We study seven kernels and two applications. *DCT, FFT* and *Filterbank* are traditional streaming benchmarks from the StreamIt benchmark suite [21]. These are compute-intensive applications. Other kernels, such as *Scalar Product*, *Black Scholes* and *Vector Add*, are from the CUDA SDK [22] and are mostly memory copy-intensive. *Monte Carlo*, also from CUDA SDK, is compute-intensive. *SRAD* and *CFD* are two applications from the Rodinia benchmark suite [23] and both are memory copy-intensive. To explore out-of-card situations, the benchmarks were run with larger datasets than provided in the benchmark suites. The baseline, non-pipelined translation from OpenMPC is used as the comparison point.

We used an NVIDIA Tesla M2090 GPU for our experiments. The device has 16 Streaming Multiprocessors (SMs) and remarkably large 6GB of DRAM. The GPU is connected via an x16 PCIe link to a host system consisting of an AMD Opteron Processor 6282 with 16 cores, running at 2.6 GHz. The host system has 64GB RAM. Up to 4 GPUs were connected to the host using the same PCIe bus.

We evaluate the contributions of our system in the following manner : (a) We demonstrate the ability of our system to handle large out-of-card data sizes by performing *COSP*. We compare the scalability of our approach against hand-written CUDA and baseline OpenMPC programs. (b) We evaluate the efficacy of our tuning method by comparing its performance to a naive compiler-only strategy. We also measure the overheads incurred by the tuning system. (c) To evaluate the benefits of pipelining and multi-GPU code generation, we compare the results obtained by these techniques over baseline OpenMPC.

6.1 System Scalability

COSP allows large, out-of-card data sizes to run on GPUs with limited device memories. Table 1 shows the system scalability of the *COSP* approach for three different representative benchmarks and also compares the results with hand-written CUDA and baseline OpenMPC codes. We calculate the maximum achieveable speedup from pipelining and compare it with the achieved speedup. To generate these results, we used hand-written CUDA codes from the CUDA SDK. Asynchronous transfers between the CPU and GPU require the corresponding memory to be allocated in 'pinned' pages i.e. the OS pages that can not be swapped out of the host memory. This can be achieved using the CUDA *cudaHostAlloc* API call. Since OS pages allocated in this fashion improve the overall application performance, the hand-written and base OpenMPC-generated CUDA codes were modified, allocating the host memories using *cudaHostAlloc*, to produce consistent comparison results with the CUDA versions generated by our system.

The scalability problems faced by the hand-written CUDA programs and the base OpenMPC-produced programs can be clearly seen from Table 1, since whenever the data size goes out-of-card, both these codes fail. For hand-written CUDA programs, as seen for the *Monte Carlo* benchmark, the number of *ThreadBlocks*, or the grid size, launched by the code crosses the CUDA-imposed limit of 65536 in a single dimension and the code starts failing even when the memory space requirement is sufficiently small to fit in the device. This is a severe programmability issue, as the programmer must consider all possible input sizes and manage the grid formation accordingly. A high-level programming model, such as OpenMPC, can easily tackle this issue by launching two dimensional grids if one dimension exceeds the limit. Hand-written CUDA codes underperform the baseline OpenMPC-generated codes as the input sizes grow large for the *Black Scholes* and *Monte Carlo* benchmarks. In the *Black Scholes* hand-written CUDA code, the launched grid size is constant and is better suited for small data sizes. In the *Monte Carlo* hand-written CUDA code, a part of the parallel loop is left out from the kernel and is instead run on the CPU. Our system translates this entire loop for GPU execution. Further, the *constant* memory allocation in the hand-written *Monte Carlo* CUDA code is proportional to the data size, and *constant* memory can run out of space if the problem size increases. High-level programming models can alleviate these programmability issues, ensuring the scalability of programs for arbitrary data sizes.

Performance benefits of our pipelining scheme can be seen in Table 1. This table also dispays the effectiveness of our implementation by comparing the ideal speedup that can be achieved from pipelining against the one obtained by our system. Except for the smallest data sizes, pipelining speedup numbers closely follow the ideal speedups for the *Monte Carlo* and *Scalar Product* programs. The differences between the ideal and achieved speedups are mainly due to the underperformance of the bidirectional transfer overlaps on the PCI Express (PCIe). This effect becomes a performance limiter for programs like *Black Scholes*, which have large bidirectional data transfers. For the small dataset of *Monte Carlo*, the optimal SplitSize suggested by our tuning system was 8192, being just one fourth of the input iteration space, thereby lowering the obtained pipelining benefits. Similar is the case for *Black Scholes*. Constant speedups for any large data size demonstrate the scalability of our approach.

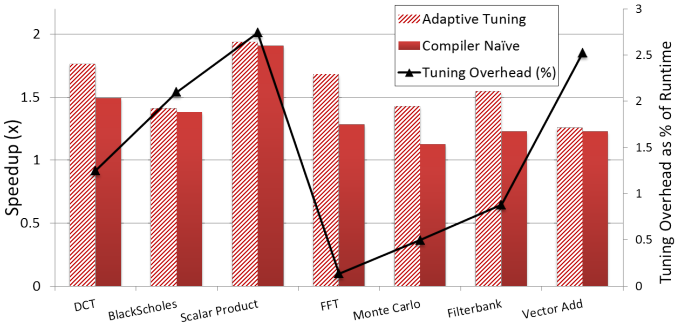

Figure 8: Performance of Adaptive Runtime Tuning System : Speedups are with respect to the OpenMPC non-pipelined baseline. Higher performance of the tuned program versions over a naive pipelining approach emphasize the necessity of tuning.

6.2 Tuning System Performance

To measure the performance benefits gained from tuning the pipelined system, we compare the speedups achieved by our adaptive tuning method over the baseline OpenMPC codes against a *naive* compiler-driven splitting strat-

Benchmark	DataSize (Iteration Space)	CUDA Time (s)	OpenMPC Base Time (s)	%Copy-in Time(s)	%Copy-out Time(s)	%Kernel Time(s)	OpenMPC Pipelined Time (s)	Speedup Ideal	Speedup Achieved
Scalar Product	1024 x 1024	0.633	0.88807	52.19789	0.22676	47.57535	0.46297	1.91579	1.91822
	1024 x1024 x2	1.267	1.7723				0.91496		1.93703
	1024 x1024 x4	----	----				1.73067		
Monte Carlo	1024 x32	0.00537	0.00454	21.55109	13.71958	64.72933	0.0037	1.54489	1.22733
	1024 x1024 x32	***	1.79902				1.24699		1.44268
	1024 x1024 x64	***	3.5924				2.51465		1.42859
	1024 x1024 x128	----	----				5.02565		
Black Scholes	1024 x16	0.00105	0.00158	46.64777	41.9736	11.37863	0.00164	2.14373	0.96344
	1024 x1024 x128	1.8598	1.22591				0.87698		1.39786
	1024 x1024 x256	----	2.457				1.73985		1.41219
	1024 x1024 x384	----	----				2.60183		

Table 1: Scalability of the *COSP* and Pipelining Mechanism : We compare the execution times of hand-written CUDA, baseline OpenMPC and compute split, pipelined OpenMPC programs. The ideal speedup is calculated using Eq. 2. In the table, '*' represent failure of the code due to larger-than-allowed grid sizes used. '—' represent code failure due to out-of-memory data size errors. Scalability of our approach can be gauged as arbitrarily large problems with out-of-card data sizes can be run and the speedup achieved for any large data size remains almost constant.**

egy. We found 1024 splits to be a good number that generated performance improvements and chose it as the "naive" reference point.

Fig. 8 displays the effectiveness of the adaptive runtime tuning system over this naive strategy. The superior performance of the tuned codes over the naive strategy indicates the importance of tuning; a static estimate of the number of splits can not provide the best performance. Further, a number of splits that yields good results for a given program may be suboptimal for another program. We also measure the overheads incurred by the adaptive runtime tuning system in terms of the percentage runtime spent in tuning. Because the tuning system runs only a single Split of the Split Loop per tuning configuration, the runtime overhead is low. The maximum overhead, measured as the percentage runtime of the total execution time, is less than 3%. Note that the tuning overhead decreases as the computation size grows.

6.3 Overall Performance Comparison

We now present comprehensive results over all benchmarks showing the effects of pipelining on 1, 2 and 4 GPUs in Fig. 9. Since the maximum speedup over the baseline system that can be achieved with pipelining is limited by three in our setup, the theoritical maximum speedup that can be achieved by pipelining, implemented on four GPUs, can be at most twelve.

Benchmarks like *DCT, FFT, Scalar Product, Monte Carlo, Filterbank* show large performance benefits since they have balanced computation and memory transfer contents, translating into equally sized pipeline stages. *SRAD, CFD and Vector Add* have a low computation-to-communication ratio and therefore show lesser overall speedups. Highly compute-intensive benchmarks like *Filterbank, Monte Carlo* show excellent scalability when multiple devices are used.

Secondly, not all benchmarks show large performance benefits when run with multiple GPUs; the reason being the bottleneck formed on the PCIe bus while transferring the data.

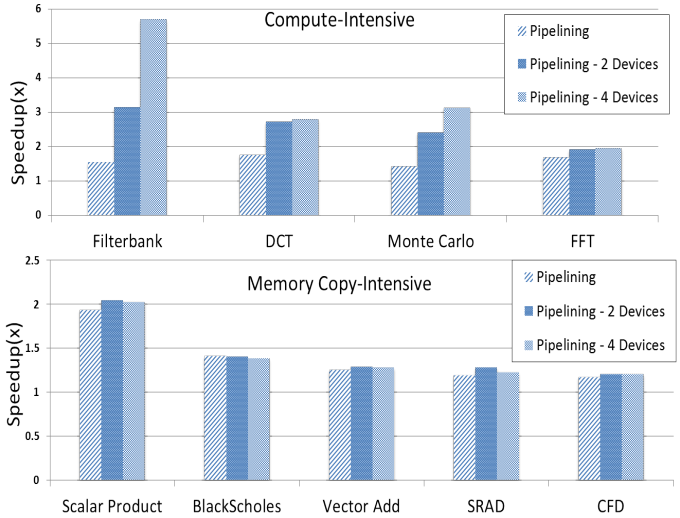

Figure 9: Speedup over the Baseline OpenMPC Generated Codes (without pipelining) : Compute-intensive applications show good scalability with multiple GPUs. Memory copy-intensive programs scale poorly with multiple GPUs since PCIe bus forms a bottleneck.

7. RELATED WORK

An alternative to deal with problems of arbitrarily large data sizes can be to write the basic program flow in terms of a stream graph. Each kernel can then be made to have a granularity of a single iteration of the stream graph. Two recent systems propose mechanisms to convert streaming programs written in StreamIt [24] language into GPU codes. The first system, Sponge [25], suggests mechanisms to optimize the stream graph. It then splits the graph into multiple kernels. Huynh et al. propose another system [26] that takes a different approach and places an entire iteration of a stream graph on a single Streaming Multiprocessor (SM). This mechanism can face scalability issues due to the restric-

tion of running just one iteration on an SM. Since each filter in a StreamIt graph consists of its own inputs and outputs, global memory accesses can be overwhelming. Both these systems therefore propose mechanisms to prefetch data into the GPU shared memory and optimize the shared memory usage. However, the behavior of these systems is undefined if the data size required is larger than the GPU memory. Secondly, the applicability of the stream programming model to large applications is still an open challenge. Further work by Huynh et al. [27] performs multi-level partitioning of the stream graph to overcome the scalability challenges in [26]. In this work, a mechanism to port StreamIt programs to multi-GPUs on the same system has also been proposed. This system heterogeneously executes kernels on several GPUs, owing to the outcome of the multi-level partitioning performed on the StreamIt graph. By contrast, our system partitions the work homogeneously amongst GPUs and yields the best possible pipelining code with the help of a fast tuning system.

A naive CUDA-provided solution to deal with out-of-card data sizes is to make use of *Zero Copy* memory. The *Zero Copy* memory mechanism allocates the corresponding buffers in the host memory instead of the device memory. The pointers to the buffers on the CPU and the GPU are mapped to each other. GPU reads the memory objects directly from the host during the kernel execution. Therefore, if the kernel has significant amount of reuse, the overheads of copying from the CPU memory would degrade the performance.

Pipelining benefits have been previously explored for GPUs to hide the global memory access latencies by generating a software pipeline that copies data into shared memory [28, 29, 26, 25]. Aji et al. propose an approach [30] that deals with a network of nodes containing GPUs, aiming to overlap the communication between nodes with computations. Our work complements these techniques since our goal is to efficiently pipeline the CPU-GPU bus.

GPU performance tuning has been a research challenge; many approaches try to model and tune the GPU performance. While many proposed tuning systems are specific to the problem domains (3D Stencil [31], N-body simulations [32], 3-D FFT [33], SpMV [34]) , Ryoo et al. [15] propose generic performance metrics that help pre-select some potential parameter configurations to choose from. A distinguishing factor of our work is the tuning objective: In contrast to approaches that try to optimize the size of the *ThreadBlock*, our work attempts to optimize the grid size i.e. the number of *ThreadBlocks*. OpenMPC's inherent tuning system performs compiler-driven optimization option pruning to generate a set of parameter configurations. However, both the OpenMPC tuning system and the space pruning proposed by Ryoo et al. are run offline and require multiple complete runs of the entire programs to choose the best configuration.

Partitioning the application so as to make use of multiple GPUs attached to the same host CPU is an emerging research challenge. While some approaches target specific applications, e.g. Matrix multiply [7], fluid simulations [8], Kim et al. [35] provide an OpenCL based approach that provides a single device image of a multi-GPU system to the application developer. However, the scalability of this approach is bound by the total GPU memory sizes.

By contrast, ours is the first work that automatically deals with out-of-card data sizes and generates pipelined, multi-GPU code for generic applications, starting with a high-level program representation.

8. CONCLUSION

We have described a novel pipelining optimization that is able to scale large-data computations on multi-GPU accelerators. To this end, the technique splits a computation so it fits in the available memory resources and overlaps data transfer with computation. The execution can take advantage of multiple GPU devices. We have demonstrated that the technique can successfully execute out-of-card datasets that would fail without our optimization. For programs and datasets whose baseline versions succeed, our technique improves performance by 1.49x, on average, owing to the computation-communication overlap. We have implemented our optimization in one of the most advanced compilation platforms for GPGPUs: OpenMPC. This system converts OpenMP programs to CUDA, performing several advanced transformations. OpenMPC provided a state-of-the-art baseline for our measurements; it also allowed us to demonstrate our technique in the context of an important current language trend, which is the extension of OpenMP with accelerator directives.

9. ACKNOWLEDGMENTS

This work was supported, in part, by the National Science Foundation under grants No. CNS-0720471, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

10. REFERENCES

[1] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. "Automatic CPU-GPU communication management and optimization". *SIGPLAN Not.*, 47(6):142–151, June 2011.

[2] Seyong Lee and Rudolf Eigenmann. "OpenMPC: Extended OpenMP Programming and Tuning for GPUs". In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society, 2010.

[3] Mehdi Amini, Fabien Coelho, Francois Irigoin, and Ronan Keryell. "Static Compilation Analysis for Host-Accelerator Communication Optimization". In *24th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, sep 2011. Also Technical Report MINES ParisTech A/476/CRI.

[4] Liang Gu, Jakob Siegel, and Xiaoming Li. "Using GPUs to compute large out-of-card FFTs". In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 255–264. ACM, 2011.

[5] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. "Mars: a MapReduce framework on graphics processors". In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 260–269. ACM, 2008.

[6] J.A. Stuart and J.D. Owens. "Multi-GPU MapReduce on GPU Clusters". In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1068 –1079, may 2011.

[7] Fengguang Song, Stanimire Tomov, and Jack Dongarra. "Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems". In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 365–376. ACM, 2012.

[8] Fengquan Zhang, Lei Hu, Jiawen Wu, and Xukun Shen. "A SPH-based method for interactive fluids simulation on the

multi-GPU". In *Proceedings of the 10th International Conference on Virtual Reality Continuum and Its Applications in Industry*, VRCAI '11, pages 423–426. ACM, 2011.

[9] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2011.

[10] T.D. Han and T.S. Abdelrahman. "hiCUDA: High-Level GPGPU Programming". *Parallel and Distributed Systems, IEEE Transactions on*, 22(1):78 –90, jan. 2011.

[11] Sain zee Ueng, Melvin Lathara, Sara S. Baghsorkhi, and Wen mei W. Hwu. "CUDA-Lite: Reducing GPU programming complexity". In *In: LCPCâĂŹ08. Volume 5335 of LNCS*, pages 1–15. Springer, 2008.

[12] Y. Yan, M. Grossman, and V. Sarkar. "JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA". *Euro-Par 2009 Parallel Processing*, page 887âĂŞ899, 2009.

[13] OpenACC. `http://www.openacc-standard.org/`, November 2011.

[14] Sunpyo Hong and Hyesoon Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness". In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 152–163. ACM, 2009.

[15] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. "Program optimization space pruning for a multithreaded gpu". In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 195–204. ACM, 2008.

[16] NVIDIA Corporation. *CUDA C Best Practices Guide*, May 2011.

[17] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. "OpenMP to GPGPU: a compiler framework for automatic translation and optimization". *SIGPLAN Not.*, 44(4):101–110, February 2009.

[18] OpenMP. Improvement to support Accelerators `http://openmp.org/wp/2012/03/openmp-is-being-improved-for-accelerators-multicore-and-embedded-systems/`, 2012.

[19] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel Midkiff. "The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation". *International Journal of Parallel Programming*, pages 1–15, 2012. 10.1007/s10766-012-0211-z.

[20] Hansang Bae and Rudolf Eigenmann. "Interprocedural symbolic range propagation for optimizing compilers". In *Proceedings of the 18th international conference on Languages and Compilers for Parallel Computing*, LCPC'05, pages 413–424. Springer-Verlag, 2006.

[21] StreamIt Benchmarks [Online]. Available: `http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml`.

[22] NVIDIA. GPU Computing SDK [Online] `https://developer.nvidia.com/gpu-computing-sdk`.

[23] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. "Rodinia: A benchmark suite for heterogeneous computing". In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54. IEEE Computer Society, 2009.

[24] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. "StreamIt: A Language for Streaming Applications". In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196. Springer-Verlag, 2002.

[25] Amir H. Hormati, Mehrzad Samadi, Mark Woh, Trevor Mudge, and Scott Mahlke. "Sponge: portable stream programming on graphics engines". In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 381–392. ACM, 2011.

[26] A. Hagiescu, Huynh Phung Huynh, Weng-Fai Wong, and R.S.M. Goh. "Automated Architecture-Aware Mapping of Streaming Applications Onto GPUs". In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 467 –478, may 2011.

[27] Huynh Phung Huynh, Andrei Hagiescu, Weng-Fai Wong, and Rick Siow Mong Goh. "Scalable framework for mapping streaming applications onto multi-GPU systems". In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 1–10. ACM, 2012.

[28] M. Bauer, H. Cook, and B. Khailany. "CudaDMA: Optimizing GPU memory bandwidth via warp specialization". In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1 –11, nov. 2011.

[29] Abhishek Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. "Software Pipelined Execution of Stream Programs on GPUs". In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 200–209. IEEE Computer Society, 2009.

[30] A.M. Aji, J. Dinan, D. Buntinas, P. Balaji, Wu chun Feng, K.R. Bisset, and R. Thakur. "MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-based Systems". In *High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on*, pages 647 –654, june 2012.

[31] Yongpeng Zhang and Frank Mueller. "Auto-generation and auto-tuning of 3D stencil codes on GPU clusters". In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 155–164. ACM, 2012.

[32] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. "Scaling Hierarchical N-body Simulations on GPU Clusters". In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11. IEEE Computer Society, 2010.

[33] Akira Nukada and Satoshi Matsuoka. "Auto-tuning 3-D FFT library for CUDA GPUs". In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10. ACM, 2009.

[34] Jee W. Choi, Amik Singh, and Richard W. Vuduc. "Model-driven autotuning of sparse matrix-vector multiply on GPUs". In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126. ACM, 2010.

[35] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. "Achieving a single compute device image in OpenCL for multiple GPUs". In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 277–288. ACM, 2011.