SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

March 2012 Vol. 55 No. 3: 663–676
doi: 10.1007/s11432-011-4497-z

CUDA-Zero: a framework for porting shared memory GPU applications to multi-GPUs

CHEN DeHao*, CHEN WenGuang & ZHENG WeiMin

Department of Computer Science and Technology, Tsinghua University, Beijing 100084*, China*

Received January 31, 2011; accepted May 23, 2011

Abstract As the prevalence of general purpose computations on GPU, shared memory programming models were proposed to ease the pain of GPU programming. However, with the demanding needs of more intensive workloads, it's desirable to port GPU programs to more scalable distributed memory environment, such as multi-GPUs. To achieve this, programs need to be re-written with mixed programming models (e.g. CUDA and message passing). Programmers not only need to work carefully on workload distribution, but also on scheduling mechanisms to ensure the efficiency of the execution. In this paper, we studied the possibilities of automating the process of parallelization to multi-GPUs. Starting from a GPU program written in shared memory model, our framework analyzes the access patterns of arrays in kernel functions to derive the data partition schemes. To acquire the access pattern, we proposed a 3-tiers approach: static analysis, profile based analysis and user annotation. Experiments show that most access patterns can be derived correctly by the first two tiers, which means that zero efforts are needed to port an existing application to distributed memory environment. We use our framework to parallelize several applications, and show that for certain kinds of applications, CUDA-Zero can achieve efficient parallelization in multi-GPU environment.

Keywords CUDA, parallelization, data access pattern, multi-GPU

Citation Chen D H, Chen W G, Zheng W M. CUDA-Zero: a framework for porting shared memory GPU applications to multi-GPUs. Sci China Inf Sci, 2012, 55: 663–676, doi: 10.1007/s11432-011-4497-z

1 Introduction

Graphic processing unit (GPU) is increasingly used for general purpose applications. Its significant computation power can impact the applications, especially scientific computations. At the same time, people are pursuing using multiple GPUs to further improve performance, which is motivated by two reasons:

• It's always desirable to enlarge the workload or make it finer grained to achieve better precision. Though 1Tflops has already been achieved in one GPU, it's far away from enough for many scientific computations. Thus scaling it to multiple devices is a good choice.

• The total memory capacity in one GPU device is too limited. For example, in matrix multiplication, the size of the matrix cannot exceed 16k if the GPU device has 6Gbyes of device memory. It's necessary to scale to multiple GPUs if larger workload is required.

*Corresponding author (email: danielcdh@gmail.com)

There are three categories of multiple GPUs: multiple GPUs in one node; multiple nodes with one GPU in each node; multiple nodes with multiple GPUs in each node.

One commonly adopted approach to program multi-GPUs is to rewrite the application using message passing libraries such as MPI, and port the computation intensive parts of the application to GPU. This approach is preferable for the second category of multi-GPUs, i.e. a cluster of multiple nodes with one GPU installed in each node. However, it incurs a great amount of effort to forge an efficient program because both steps (writing MPI programs and writing efficient GPU programs) are not trivial. And when it develops to the third category, i.e. a cluster of multiple nodes with multiple GPUs in each node, extra effort is required to further optimize the program. NAMD [1] is a good example of this approach, which takes several skilled programmers more than 6 months of effort to write the MPI version of the program, another 3 months to port computation intensive kernels to GPU, and another 1 month to port to multiple GPUs.

An alternative approach is to start from an existing GPU implementation, and parallelize it to multi-GPUs. This approach is effective for certain kinds of applications. Unfortunately, the parallelization work is tedious, because

• Multi-GPUs are controlled using multi-threading within a computation node, which programmers need to be aware of. When developed to multiple computation nodes, programmers also need to take care of the inter-node communication. Thus it's required that the control structure of the program be modified manually.

• Currently, most programming models of single-GPU adopts shared-memory model. For multi-GPUs, each GPU has its own memory space, which can only be accessed by itself. Thus it's a distributed memory system, and programmers need to partition the workload data manually.

As observed from many prior works [2], it is already difficult to write programs for even single GPU. Programmers need to design delicate algorithms to distribute workloads to massively parallel processors. Porting single-GPU programs to multi-GPUs incurs another round of workload distribution, which not only adds extra burden to programmers, but also tends to confuse programmers when they manage two levels of workload distribution. Thus it is intuitive to design an approach to "reuse" the effort programmers have paid during the composition of shared-memory single-GPU programs, and use compiler transformations to generate multi-GPU enabled version automatically. This intuition motivated us to build CUDA-Zero, a framework to automate the parallelization for multi-GPUs. Specifically, we choose CUDA [3] as the programming model for our input single-GPU programs, because it is so far the most widely adopted programming model for GPU computing. However, our methodology can be easily adopted by other GPU programming models such as OpenCL [4] and Brooks [5].

Our framework is named CUDA-Zero because for certain kinds of CUDA applications, zero effort is required for programmers to modify the source code. The compiler will take care of all the parallelization work automatically. Additionally, annotation schemes are designed for programmers to specify data access patterns and gain better control of the parallelization process. However, experiments show that most of the time, user annotation is unnecessary. We also reason about why fully automatic parallelization is possible for certain kinds of GPU applications. Currently, our implementation is applicable for parallelization on the first category of multi-GPUs, i.e. multiple GPUs in on node. We'll discuss the possibilities of adopting CUDA programming model to other two categories. It's also applicable for the application written in MPI+CUDA for the second category multi-GPUs to upgrade to the third category. This paper has made the following contributions:

• To the best of our knowledge, it's the first work to fully automate the process of porting shared-memory GPU applications to multi-GPUs.

• It's the first attempt to utilize profiling to derive the access patterns of shared-memory SPMD kernels. A highly efficient profiling mechanism is proposed on GPU.

The rest of the paper is organized as follows: Section 2 discusses the related work. Section 3 gives an overview of the CUDA-Zero framework. During the automatic parallelization process, the key problem is how to get the access patterns of each kernel function to derive efficient data distribution schemes. In Section 4, we introduce the three-tier mechanism we adopt to get access patterns. Section 5 evaluates

the effectiveness of CUDA-Zero and analyzes the performance of several auto-parallelized applications. Section 6 concludes the paper.

2 Related work

Using GPUs for general purpose computing has enabled a range of problems to be solved faster and has allowed programmers to work on larger data sets [2]. Although most prior work has focused on applications on single GPU, there are a number of efforts studying how to utilize multiple GPUs to accelerate specific problems. Quintana-Orti et al. proposed a framework to solve dense linear systems on multi-GPU platforms [6]. Schaa and Kaeli studied the design space for multi-GPUs and proposed a performance model [7]. To scale GPU applications to larger workloads, Sundaram et al. proposed an approach [8], but it's designed for domain-specific templates. To improve the programmability of multi-GPU platform, many programming models were proposed: Moerschell and Owens implemented a distributed shared memory system to simplify execution on multiple GPUs [9]. Fan et al. explored how to utilize distributed GPU memories using object oriented libraries [10]. Strangert et al. proposed a new programming model to enable programmers to specify work distribution in a unified way [11]. All these programming models either incur too much communication overhead, or require manual specification of the workload distribution. Kim et al. built a source to source compiler to transform the OpenCL programs to multiple GPUs [12], which is similar in our work except that they take the OpenCL as the input, while we transform CUDA applications.

Parallelizing compilers such as [13] addressed the issue of automatic data partitioning for distributed memory systems. Ref. [14] derived reference patterns from array subscripts of shared memory programs, and uses pattern matching to diminish communication. Ref. [15] proposed to use constraint based method to derive optimal data partition schemes. All these studies were based on Fortran-like language, and it cannot be applied to C-like programming languages such as CUDA. This is because: (1) Most of these works assume regular control/data structure (perfect nested loop, affine loop index, etc.) However, structures such as branching and pointer referencing, which are commonly seen in C-like languages, will force these approaches to make conservative decisions that are opt to be inefficient. (2) C-like languages enable programmers to index arrays in a flattened way; thus the compiler may not have the dimensional information of the array, which is vital to the previous approaches. As a result, although SPMD kernels could be converted to nested loops [16], it's not trivial to apply the traditional auto-parallelization methods to it.

Because data partition is difficult to perform automatically, several programming models [17–20] were proposed to help programmers annotate on source code to specify data partition. We borrowed a subset of these annotations, and applied them to CUDA-Zero. To enable programmers to have more control on their programs, we also showed that most of the time, optimal data partition can be acquired automatically by compilers. Additionally, programming models such as MapReduce are proposed on GPU [20], and also ported to multi-GPUs [21]. Though it's convenient to write programs following such programming models, it's difficult to write highly efficient programs because it abstracts too much details from the hardware layer.

3 CUDA-Zero

CUDA is Nvidia's shared-memory programming model for GPU. Ref. [2] gives a comprehensive introduction to CUDA architecture and programming methodology. In CUDA, programmers specify two constructs: kernel functions and host functions. Kernel functions are run on GPU device in the SPMD way, i.e. each device can have thousands to millions of threads running the same kernel function in parallel. Host functions are run on host machines, which invoke kernel functions by specifying the thread count. Each thread owns a unique index for identification, which is used to specialize the execution of the kernel function. Thread block is a collection of threads which can perform barrier synchronization.

Threads between different thread blocks cannot synchronize. Thus thread blocks are the basic building blocks which are independent of each other. The thread count specified by the host function is further divided into two dimensions: $\mathrm{DIM}_{\mathrm{block}}$ (number of threads in each thread block) and $\mathrm{DIM}_{\mathrm{grid}}$ (number of thread blocks). Either of these dimensions can have 3 sub-dimensions: x, y and z. As for the massively parallel nature of CUDA, it's required that there should be many threads (usually 64–256) in each thread block, and there are enough thread blocks (usually more than 100) to make all computation units fully occupied. Thus one of the most important concerns when writing CUDA programs is to set the granularity fine enough to generate a large number of threads. Because thread block is an independent unit, it's common practice that one only needs to increase the number of total thread blocks when the workload scales.

The CUDA-Zero compiler is a source to source compiler built on top of the Cetus project. It takes CUDA program as input, performs a series of transformations on it, and outputs another CUDA program, which is enabled with multi-GPU support. The transformations include the following parts.

3.1 Workload distribution and threading

Because the CUDA programmers already paid the effort to divide the workload into threads, we can reuse the information of thread division. A job is referred to as a group of threads that's scheduled to run as one kernel invocation in a GPU device. Each kernel invocation in the original CUDA program is divided into several jobs, which can be invoked in distributed GPU devices. Intuitively, as the communications between different GPUs are expensive, we choose to divide the workload at the granularity of thread block to make each job independent of each other. If there is more than one sub-dimension at the thread block level, we pick one dimension that incurs the least overhead (detailed in Subsection 3.2) as the basic scheduling unit. For example, if dimension y is chosen, only the thread blocks with different block Idx.y values could be assigned to different jobs. For the convenience of illustration, in the rest of the paper, we refer to this basic scheduling unit as "block unit".

Both static and dynamic mechanisms are implemented for workload distribution. The static approach divides the workload into the number of jobs that is equal to the total number of GPUs available. Each device thread is assigned one job and has equivalent amount of workload to work on. While guaranteeing coarse granularity for each job, this approach works well for most GPU programs. However, we observed some cases where workload becomes imbalanced. Thus dynamic distribution is also implemented, which divides the workload into finer grained jobs to form a job pool. Each device thread picks up a job as soon as it is idle. Programmers can specify whether to use dynamic workload division as a compiler parameter.

The threading mechanism works as follows: the main thread spawns a certain number of device threads, each of which is attached to a GPU device. Once the GPU finishes execution, the corresponding device thread becomes idle.

3.2 Data decomposition

Once the workload is distributed into jobs, it's important to locate the memory footprint of each job. The compiler identifies for each kernel the reference set and write set of each job and decomposes the input/output data accordingly. This is done by analyzing the access pattern of each array in the kernel functions. This problem has been addressed by a good many prior works [13–15, 17–19], and is proved to be difficult. Most prior approaches either suffer from too much communication overhead, or require heavy user annotation to tell the compiler how to partition the data. However, our observations show that certain kinds of CUDA applications have very regular data access patterns, which is easy to model and possible to recognize automatically.

There are four kinds of memories in CUDA architecture. We derive access patterns for global memory and texture memory. For shared memory, as it's only used as temporary space within a thread block, there's no need for decomposition. As constant memory is very small, it's reasonable to replicate it among all jobs. For arrays in global memory, we further divide them into input arrays, output arrays, and input/output arrays. Observations show that for output arrays and input/output arrays, the write-sets

are distinctive among different jobs. This can be reasoned by the fact that there's no inter-thread-blocks synchronization, and thus there shouldn't be any output dependency between different thread blocks. As a result, it's always viable to decompose output arrays and input/output arrays into distinctive sub-arrays. However, for input arrays, there could be data sharing between different block units. The worst case is when each block unit accesses all the array elements. As a result, it's sometimes necessary to replicate the input arrays.

Because arrays are usually passed in kernel functions as pointers, the dimensional information is lost. As a result, array accesses are mostly presented in a flattened way. The term "access pattern" is with respect to a selected sub-dimension, i.e. the block units are pre-determined. Access pattern has two meanings:

Distribution: For a specific block unit, how the accessed elements are distributed in the entire array.

Continuation: The relationship of the accessed elements between consecutive block units.

We abstract the access patterns into 3 categories:

• Contiguous (n): All the accessed elements for a specific block unit are distributed in a contiguous index range with a length of (n); the index ranges for consecutive block units are also consecutive, i.e. the first element of the next block unit immediately follows the last element of the previous block unit.

• Cyclic(n_1, n_2, n_3): The array can be divided into n_3 consecutive sub-arrays. In each sub-array, the access pattern follows Contiguous(n_1); the distance between two consecutive sub-arrays is of the value n_2.

• Unknown: Besides the above two access patterns, we consider it as unknown.

As shown in [17], the Unknown pattern can be further divided into more varieties. We only choose these three categories for simplicity. As observed from most CUDA applications we've seen so far, the first two patterns serve to represent typical CUDA access patterns. However, there're some complex patterns such as "Blocked", "Overlapped Contiguous" that we haven't studied. We'll leave them as the future work.

The data decomposition determines the portion of data to be stored in each device. Having derived the access pattern, the data to be stored is fixed to those that could be accessed by the job. Assume there are S block units in each job, and there are T block units in total. For Contiguous(n) access pattern, a memory area of $n * S$ contiguous elements is copied for each job; for Cyclic(n_1, n_2, n_3) access pattern, a loop is formed to iterate across the array, copying $(n_1 * S)$ in each iteration, with the step of n_2 and trip count of n_3; for Unknown access pattern, we replicate the whole array for every GPU device. In Contiguous (n) access pattern, if total size of the array is known, because the amount of data accessed by each block unit is equal, it's unnecessary to acquire the parameter n for data decomposition. We can use size_of_array$\times\frac{s}{\tau}$ to derive the data needed to be copied for each job.

Having derived the data set to be stored in each device, the next step is to find the places to insert memory copies to transfer the data. The memory copies in the original single-GPU CUDA programs should be adjusted to the new data decomposition. If multiple kernels exist, additionally memory copies of an array A should be added after a kernel invocations K if there is at least one kernel K' invocation in the transitive closure of K's successors, and the following two attributes satisfy for K':

• There's flow dependency on the array A from K' to K.

• The access pattern on array A is different between K and K'.

For some cases, additional memory copies added after a kernel invocation can be extremely costly, especially when the kernel is inside a loop. Following [7], a performance model is built to determine whether the benefit of parallelizing the kernel could offset the overhead introduced by the additional memory copies. If the overhead is too much, the workload of the kernel is replicated instead of being distributed to every device.

To model the overhead of data transfers for a specific kernel, some observations are considered: duplicating the whole array is the most costly approach because it requires the maximum data transfer. Using loop to copy cyclic data from/to GPU device is less costly than duplication, but still incurs more overhead than copying contiguous data because each memory copy incurs a certain amount of overhead. The overhead model of a kernel K is shown in formula (1), in which C is a constant derived by micro-

benchmark for a specific CUDA environment to model the overhead of each memory copy operation, and N is the total number of devices in the environment.

$$O(K) = \sum_{i \in K} S(i) \times \begin{cases} 0, & i \text{ does not need to copy}, \\ 1, & i \text{ is Contiguous}(n), \\ 1 + C/n_2, & i \text{ is Cyclic}(n_1, n_2, n_3), \\ N, & i \text{ is Unknown}. \end{cases} \tag{1}$$

The overall overhead of the program is modeled as Eq. (2), in which P is the set of all kernel invocations in the program. The tripcount is derived from execution profile if available. Otherwise, its value is set to a constant.

$$O_{\text{all}} = \sum_{k \in P} \begin{cases} O(k), & k \text{ isoutside any loop}, \\ \text{tripcount} \times O(k), & k \text{ isinside a loop}. \end{cases} \tag{2}$$

Note that the data decomposition is with respect to a chosen sub-dimension, i.e. the block units are pre-determined. As mentioned in Subsection 3.1, overall overhead is calculated for every combination of sub-dimensions for all kernel invocations. The one that incurs the least overall overhead is chosen for the workload distribution.

3.3 Array index transformation

For the decomposed workload and data, transformations of the array indexes are needed to ensure the correctness of the array accesses. This invokes two steps:

- Map the index of the block unit in each job to the original index of the undivided workload.
- Map the index of the original array accesses to the index of the decomposed arrays.

The two kinds of mappings are summarized in Table 1. Note that for Contiguous access pattern, it's unnecessary to transform the array index because two mappings are complimentary.

4 Access pattern recognition

As seen from the previous section, both data decomposition and array index transformation require the knowledge of array access pattern. In this section, we propose a three-tier approach to derive this information.

4.1 Static analysis and pattern matching

Scalar expansion is performed to transform each array access into a function of block indexes, thread indexes, formal arguments of the kernel function, induction variables (if applicable) and constant values. This function is abstracted to build a model, in which the factor of block indexes and thread indexes are recorded. If the array access is nested in a countable loop, the trip count of the loop is also recorded. For example, the array access expressions in Figure 1(a) is modeled as shown in Figure 1(b).

We compare the model we build for each array access to the patterns we already know. If the pattern match is successful, the access pattern is recorded. Otherwise the access pattern is recorded as Unknown. Figure 2 shows some patterns we use. If there are multiple array accesses for a same array, only when the access patterns of all the array accesses are identical can we use it as the access pattern of the array. Otherwise, the access pattern of the array is set as Unknown.

This approach is applicable for many CUDA applications. And the pattern database can be enlarged to make it suitable for more applications. However, in some situations, two major obstacles make this approach incapable:

- Complex control flow and pointer references can make scalar expansion difficult. In this situation, the access model cannot be built effectively.

Table 1 Mappings in array index transformation

	Mapping of thread block index	Mapping of the array index
Contiguous(n)	Index_{tb}+Job_idx*Job_Size	$\text{Index}_{\text{array}} - n*$ Job_Idx*Job_Size
Cyclic(n_1, n_2, n_3)	Index_{tb}+Job_idx*Job_Size	$\text{Index}_{\text{array}} - (n_2 - n_1)*$ FLOOR($\text{Index}_{\text{array}}/n_2$)
Unknown	Index_{tb}+Job_idx*Job_Size	$\text{Index}_{\text{array}}$

Expression 1(E1):
Histogram[NUM_BINS*$I_b.x$+$I_t.x$]

Expression 2(E2):
for (i=0; i<$D_b.x$; i++) {
 ...
 ...=A[$D_b.x$*$D_t.x$*$D_t.y$*$I_b.y$+$D_b.x$*$D_t.x$*$I_t.y$]
 +$D_t.y$*$I_b.x$+$D_t.x$*I+$I_t.x$];
}

(a)

	Block index			Thread index			IV	Trip count
	x	y	z	x	y	z		
E1	NUM_BINS	0	0	1	0	0	0	1
E2	$D_b.x$*$D_t.x$*$D_t.y$	$D_t.y$	0	1	$D_b.x$*$D_t.x$	0	$D_t.x$	$D_b.x$

(b)

Figure 1 (a) Sample array access expressions; (b) corresponding access model of the sample array access expressions.

Access pattern	Block index			Thread index			IV	Trip count	Dim
	x	y	z	x	y	z			
Contiguous ($D_t.x$, $D_t.x$)	$D_t.x$	0	0	1	0	0	0	1	x
Contiguous($D_t.x$*$D_t.y$*$D_b.x$, $D_t.x$*$D_t.y$*$D_b.x$)	$D_t.y$*$D_t.x$	$D_t.y$*$D_t.x$*$D_b.x$	0	1	$D_t.x$	0	0	1	y
Cyclic($D_t.x$*$D_t.y$, $D_t.x$*$D_t.y$, $D_t.x$*$D_t.y$*$D_b.x$)	$D_t.y$*$D_t.x$	$D_t.y$*$D_t.x$*$D_b.x$	0	1	$D_t.x$	0	0	1	x
Contiguous(S*$D_t.x$, S*$D_t.x$)	S*$D_t.x$	0	0	1	0	0	1	S	x
Contiguous($D_b.x$*$D_t.x$*$D_t.y$, $D_b.x$*$D_t.x$*$D_t.y$)	$D_t.y$	$D_b.x$*$D_t.x$*$D_t.y$	0	1	$D_b.x$*$D_t.x$	0	$D_t.x$	$D_b.x$	Y
Cyclic($D_t.x$, $D_t.x$, $D_t.x$*$D_b.x$)	$D_t.x$	1	0	1	0	0	$D_b.x$*$D_t.x$	N	x

Figure 2 Some access patterns used in CUDA-Zero.

• Some user optimization can make the array access patterns difficult to recognize. For example, manual loop unrolling can transform an array of Contiguous pattern into several array accesses that doesn't match the Contiguous pattern.

Because of the above two obstacles, this approach only makes the most conservative estimations. To achieve more progressive estimation, we propose using execution profile to guide analysis.

4.2 Profile based analysis

Profile based analysis is used for all arrays of which the access pattern has been recognized as Unknown by the static approach. It's constituted of 4 stages: candidate selection, instrumentation, training and analysis.

4.2.1 *Candidate selection*

To be selected as candidate of profile based approach, the access pattern of the array must be independent of input data, preventing situations where there are multiple occurrences of array accesses in different control paths, and the input data may determine which control path will be taken. To verify this property, dependency analysis is performed to check whether the array access is control-dependent on any of the formal parameters (including the dimensional variables of the kernel function). However, it's a common practice that the access pattern is data- dependent on formal parameters. Thus data dependency is not chosen as a criterion in candidate selection. Subsection 4.2.4 introduces how data dependency can be tolerated using symbolization. Note that the dependency check is only needed for arrays that have more than one occurrences of array access because single occurrence would not lead to conflicting access patterns.

4.2.2 *Instrumentation*

Both host functions and kernel functions are instrumented. In the host function, code is instrumented before each kernel invocation, recording the dimensional information and the value of each actual parameter. This information is used to derive the attributes of the access pattern in a symbolic way, which will be introduced in subsection 4.2.4. Additionally, for each array that should be inspected, two adjoint arrays, C_{lower} and C_{upper}, are allocated in the device memory, the address of which is passed into the kernel as actual parameters. Both adjoint arrays are of the same size of the original array, but of the integer type. C_{lower} is used to record the lower bound of the block unit for each element of the array, while C_{upper} is used to record the upper bound. All the elements in C_{lower} are initialized as -1, while those in C_{upper} are initialized as$(M+1)$, in which M is the maximum block index of the kernel.

In the kernel function, codes are instrumented before each access of those arrays to be inspected. Assuming the index of the accessed element is I_{current}, the instrumented code compares the current block unit index with the value of $C_{\text{lower}}[I_{\text{current}}]$, and chooses the less significant value to update $C_{\text{lower}}[I_{\text{current}}]$. Thus after the kernel invocation, the C_{lower} array stores the lower bound of block unit index that have accessed the corresponding element in the original array. Similar codes are instrumented for C_{upper} to get the upper bound. Note that it's possible that more than one block unit can access the same array elements simultaneously. Thus it's necessary to guard the compare and update operation as an atomic operation, which can be fulfilled by the CAS atomic operation provided by CUDA.

4.2.3 *Training*

During the candidate selection stage, we have already filtered out those arrays that have access patterns control dependent on input data. We also work out the problem of data dependency using symbolization (detailed in Subsection 4.2.4). As a result, the output of profile based approach is independent of the input data. Thus our profiling approach doesn't share the common problem of the profile based analysis, i.e. the representativeness of the training data. This feature makes it very easy to select data for training runs. We encourage using small workload as the training data so that in practice, training time can be negligible. We also encourage using different sets of data for multiple training runs. This is important for the analysis phase to enable the symbolized representation of the access pattern.

4.2.4 *Analysis*

The analysis phase is further divided into three tasks:

Legality test. For each candidate array, compare the adjoint arrays C_{lower} with C_{upper}. If they are not identical, there can be multiple block units accessing the same element. In this case, CUDA-Zero makes conservative assertion to set the access pattern as Unknown.

Numeric representation of access pattern. Once two adjoint arrays are proved to be identical, a series of reductions are applied to both adjoint arrays to derive the access pattern in the numeric representation. A triplet (initial, step, length) is used to represent an arithmetic progression, in which initial can be either an integer or an arithmetic progression; step and length are integers. The output of the reductions is an arithmetic progression, which is matched to existing patterns to derive the access pattern in numerical manner. Two examples of both Contiguous and Cyclic access patterns are given in Figure 3.

In the numeric representation, the parameters n, n_1, n_2, n_3 of the access pattern are represented as an integer value, which are usually dependent on the input workload. The purpose of this step is to represent these parameters in a symbolized way, so that the data dependency can be resolved and the access pattern can be generalized for all workloads. The mapping between each symbol and value, which are recorded before each kernel invocation, are combined to form a product to match the numeric representation. However, it's possible to find more than one match. Because we have already passed the legality test, at least one match should be the correct representation of the access pattern. In this case, extra training runs (with different input data) are used to verify the correctness of the matches and prune the result. The process ends until only one matching is left, or all the training data are tested. If an exclusive match

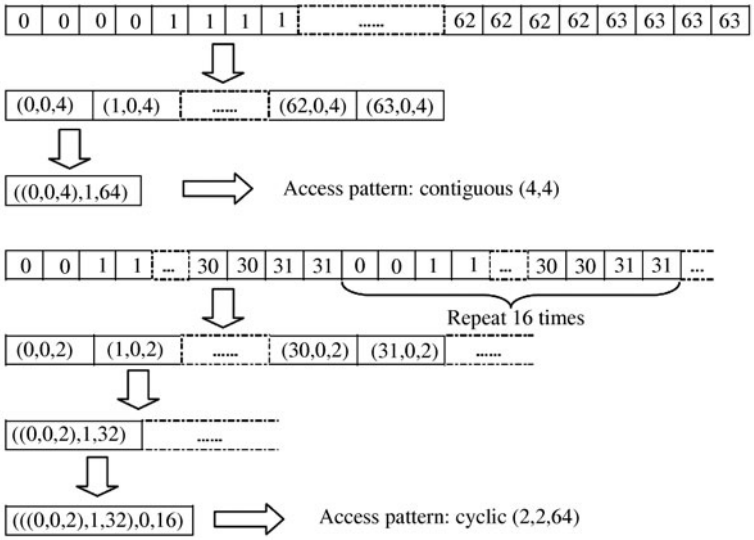

Figure 3 Reductions to derive the numeric representation of the access pattern.

is found, then it should be the correct representation of the access pattern. However, it's not always possible to find an exclusive matching. In this case, there are two possibilities: multiple matches are equivalent; or some matches are incorrect but difficult to catch. Observations show that if the multiple matches are verified by multiple training runs, it's likely that they are equivalent. However, for correctness issues, the access pattern is still set to Unknown, and programmers are given an advisory on the potential access patterns of the array, which can be used for user annotation.

Let it be noted that sometimes unnecessary to have symbolic representation of the access pattern. As described in Subsection 3.2, if the numeric representation can already prove that the access pattern is Contiguous, it suffices to perform automatic workload distribution and data decomposition.

4.2.5 *Discussion*

The profile based analysis can derive the access patterns that static analysis cannot get. For example, when there are many memory accesses to a same array in the kernel program, the static analyzer usually cannot find the right access pattern because multiple complex access patterns can combine to form a simple access pattern.

The limitation of the profile based analysis is that for certain types of complex access patterns, it cannot derive the access pattern. For example, for code with manual array padding, the profile based analysis will fail to get the access pattern. Instead, it will resort to the use of the replication to ensure the correctness.

4.3 User annotation

User annotation is adopted as a backup plan when the previous two approaches fail to find the access pattern that programmers are aware of. However, we do not find it necessary to refer to this approach during our experiments.

To annotate for access pattern, the programmers only need to specify it explicitly as pragma statements in each kernel function. A code snippet is shown in Figure 4.

5 Evaluation

We evaluated CUDA-Zero on 5 GPU benchmarks. CP, TPACF, MRI-Q, PNS are from Parboil benchmarks [22], Matrix Multiply is from NVidia SDK [3]. Our performance testing is performed on a Tesla C1060 machine constituted of 4 GPUs, each of which has 4Gbyte memory and a peak performance of 1TeraFlops per second.

```
__global__ void test (float* C, float* A, float* B) {
    #pragma dimension y
    #pragma array C continuous
    #pragma array A continuous
    #pragma array B intervene (N)
    int index_a=.....;
    int index_b=.....;
    int index_c=.....;
    C[index_c]=A[index_a]+B[index_b];
}
```

Figure 4 A code snippet of the user annotated pragmas.

Table 2 Array access pattern recognition by different approaches

	Total arrays	Static	Profile	Replcated
Matrix multiply	3	0	2	1
CP	1	0	2	0
TPACF	2	0	2	1
MRI-Q	8	8	0	0
PNS	3	2	1	0

5.1 Parallelization effort

Through our experiments, all the 5 benchmarks can be parallelized automatically without referring to any user annotation. Table 2 shows how the access patterns of arrays can be recognized by different tiers of approaches.

For all output arrays, the first two tiers of approaches can automatically derive the access pattern. For some input arrays, there can be sharing between different block units. Thus full duplication is performed. E.g. in Matrix Multiply, which calculates $A * B$, array B is accessed by all block units if dimension y is chosen to define block unit. In order to make both A and B dividable, workload needs to be distributed in a "Blocked" way, i.e. x and y dimensions should be combined to define block unit. We leave this kind of workload distribution as future work. In TPACF, the data sharing between block units is complex and unstructured; thus user intervention is needed if the data need to be further decomposed.

5.2 Performance

The performance of the parallelized benchmarks is shown in Figure 5. We use large workloads to test all performance, ensuring the total execution time is larger than 10 seconds. This is because when the execution time is small, the constant runtime of some CUDA library calls such as cudaSetDevice() will dominate the execution time. Most benchmarks we experimented can achieve near linear speedup against single GPU execution. Matrix Multiply suffers the worst speedup because one array is replicated, and thus the communication overhead of this extra array kills the speedup in great margin. For TPACF benchmark, though one array is replicated, as the size of the array is relatively small, the communication overhead is negligible.

5.3 Scalability

In the scalability test, we evaluated two factors: more GPUs, larger workloads.

5.3.1 *More GPUs*

In the previous section, we show that CUDA-Zero scales well in up to 4 GPUs. Though we haven't implemented a cluster version, we evaluated the possibility of scaling CUDA-Zero to more GPUs, which includes two major concerns: scalability of the kernel and communication overhead.

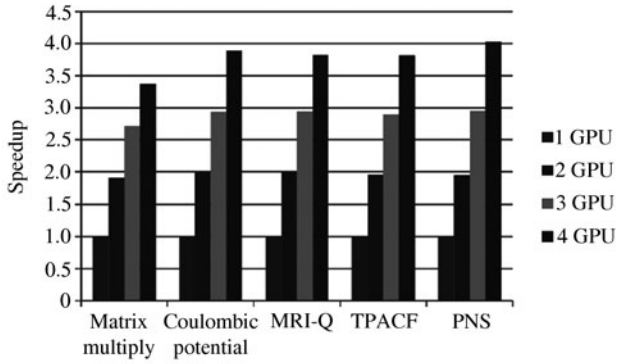

Figure 5 Performance of parallelized applications.

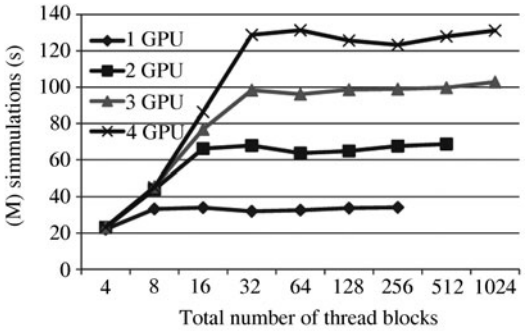

Figure 6 Performance impact of thread block number on PNS benchmark.

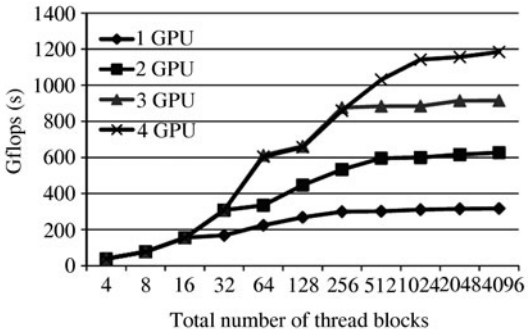

Figure 7 Performance impact of thread block number on MRI-Q benchmark.

Following our workload distribution mechanism, when the number of GPU devices increases, the total number of thread blocks in each kernel invocation will decrease. It's worthwhile to calculate the lower bound of thread blocks in each kernel invocation to achieve maximum computation power for each device. For a certain amount of workload, we manually modify the total number of thread blocks in the original CUDA application to measure the performance. Figure 6 and Figure 7 show the performance impact on PNS and MRI-Q benchmarks respectively. The performance of PNS benchmark is bounded by memory bandwidth; thus 32 thread blocks already suffice to achieve linear speedup for 4 GPUs. MRI-Q is more computation intensive; thus the speedup for 4 GPUs stays sustainable after more than 1024 thread blocks are available. In general, it's desirable to have a massive number of thread blocks to achieve scalable speedup for kernel execution.

Communication overhead varies among applications. In MRI-Q, TPACF, CP and PNS, the overhead is negligible because the communication time is small compared with computation. Matrix Multiply shows the worst communication overhead. A detailed breakdown of matrix multiplication's execution time is shown in Figure 8. In general, the communication overhead is less significant in larger workloads. For the decomposed arrays, the communication time decreases as the device count increases. But as the bandwidth of host memory and PCI-Express bus saturates, the communication time stops decreasing. However, when scaling to multiple nodes, additional host memory bandwidth and PCI-Express bus bandwidth can be achieved from the distributed nodes. Thus the communication overhead of decomposed arrays will decrease when scaling to multiple nodes. For replicated arrays, because current CUDA doesn't support broadcasting from host memory to device memory in all devices, the total amount of transferring data is in proportion to the device count. With broadcasting operation supported, better scalability can be achieved for replicated arrays.

5.3.2 *Larger workloads*

In CUDA-Zero, if all the arrays are distributed, workloads can be scaled quite well. In Figure 9, single GPU cannot handle more than 256 thread blocks simply because global memory is not enough to hold

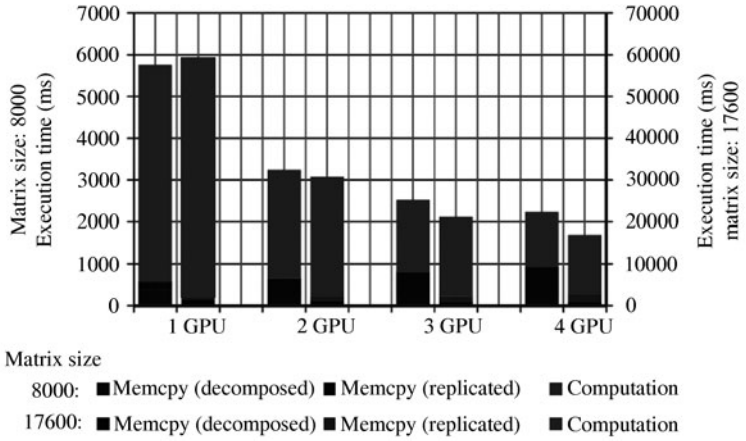

Figure 8 Execution time breakdown for matrix multiplication. The left bar presents the small workload with matrix size of 8000×8000; the right bar presents the large workload with a matrix size of 17600×17600.

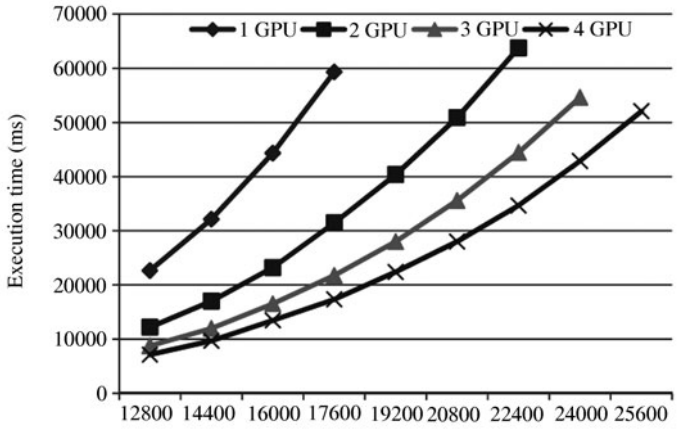

Figure 9 Workload scalability for Matrix Multiply (x-axis is the size of the array).

the data. However, with 4 GPUs, up to 1024 thread blocks can be specified simultaneously, making fully use of the memory capacity of all GPUs. For cases where some arrays are replicated, workload can still be scaled because of other decomposed arrays. As shown in Figure 9, multi-GPU version of Matrix Multiply can work on larger arrays than its single-GPU counterpart. However, it's foreseeable that for a larger system (cluster of GPUs), full decomposition is needed for every array.

5.4 Limitations

Though CUDA-Zero can achieve scalable speedup for certain kinds of CUDA application, it does have some limitations:

Atomic operation. CUDA-Zero cannot parallelize programs in which there are atomic operations accessing global memory. This is because there's no hardware support to ensure atomicity of global memory operations across different devices. And atomic operations enable different thread blocks to write to the same memory location in single kernel invocation, resulting in overlapped output set. Thus kernel invocations with atomic operations are either executed in one device or fully replicated to all devices.

Multiple kernels with irregular dependence. Some applications are comprised of multiple kernels, each of which has irregular data dependency on each other. As a result, the performance model of CUDA-Zero decides to replicate the whole workload. To parallelize this type of applications to multiple GPUs, major surgery is usually needed at algorithmic level, and thus is usually done manually. For example, in [6], a

significant amount of algorithmic effort has been made to port dense linear applications to multiple GPUs. For this type of applications, CUDA-Zero cannot generate an efficient multi-GPU version automatically. However, CUDA-Zero can provide useful information (such as data access pattern, etc.) for programmers' further diagnose.

6 Conclusion

In this paper, we have described CUDA-Zero, a compiler framework to automate the parallelization process from single GPU to multiple GPUs. A three-tier approach is proposed to derive the access patterns for automatic data decomposition. Experiments show that most applications can be parallelized automatically by our framework. With effective access pattern recognition and data decomposition, most parallelized applications exhibit good efficiency and scalability.

The reason why GPU applications can be automatically parallelized to multi-GPUs is based on the fact that there is a level of independent objects (thread block as in CUDA) in most GPU programming models, facilitating the parallelizing compilers to further deploy the parallelism. As most GPU programming models are based on shared memory model, which is easier than distributed memory model, it's an efficient approach to start from single-GPU program and use automatic parallelizing tools such as CUDA-Zero to port it to large scale systems.

References

1 Phillips J C, Stone J E, Schulten K. Adapting a message-driven parallel application to gpu-accelerated clusters. In: SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. Piscataway: IEEE Press, 2008. 1–9

2 Ryoo S, Rodrigues C I, Baghsorkhi S S, et al. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In: PPoPP'08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM, 2008. 73–82

3 NVIDIA. NVIDIA CUDA Programming Guide 2.0. 2008

4 Stone J E, Gohara D, Shi G. OpenCL: a parallel programming standard for heterogeneous computing systems. Comput Sci Eng, 2010, 12: 66–73

5 Buck I, Foley T, Horn D, et al. Brook for gpus: stream computing on graphics hardware. In: SIGGRAPH'04: ACM SIGGRAPH 2004 Papers. New York: ACM, 2004. 777–786

6 Quintana Orti G, Igual F D, Quintana Orti E S, et al. Solving dense linear systems on platforms with multiple hardware accelerators. In: PPoPP'09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York: ACM, 2009. 121–130

7 Dana S, David K. Exploring the multi-gpu design space. In: IPDPS'09: Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium. New York: ACM, 2009. 1–12

8 Sundaram N, Raghunathan A, Chakradhar S T. A framework for efficient and scalable execution of domain-specific templates on gpus. In: IEEE International Parallel and Distributed Processing Symposium. Washington DC: IEEE, 2009. 1–12

9 Moerschell A, Owens J D. Distributed texture memory in a multi-gpu environment. In: GH'06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware. New York: ACM, 2006. 31–38

10 Zhe F, Feng Q, Arie K. Zippygpu: programming toolkit for general-purpose computation on gpu clusters. In: GPGPU Workshop at Supercomputing. Washington DC: IEEE, 2009. 1–12

11 Strengert M, Muler C, Dachsbacher C, et al. CUDASA: compute unified device and systems architecture. IEEE Trans Vis Comput Gr, 2009, 15: 605–617

12 Kim J, Kim H, Lee J H, et al. 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. New York: ACM, 2011. 277–288

13 Li J. Compiling crystal for distributed-memory machines. PhD Thesis. New Haven: Yale University, 1992. 1–134

14 Li J, Chen M. Generating explicit communication from shared-memory program references. In: SC'90: Proceedings of the 1990 Conference on Supercomputing. Los Alamitos: IEEE Computer Society Press, 1990. 865–876

15 Gupta M, Banerjee P. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicom-

puters. IEEE Trans Parall Distr, 1992, 3: 179–193

16 Stratton J A, Stone S S, Hwu W M W. Mcuda: an efficient implementation of cuda kernels for multi-core cpus. In: Languages and Compilers for Parallel Computing: 21th International Workshop, LCPC 2008. New York: ACM, 2008. 16–30

17 Choudhary A, Koelbel C, Zosel M. High performance fortran: implementor and users workshop. In: SC'93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing. New York: ACM, 1993. 610–613

18 Chamberlain B L, Choi S E, Lewis E C, et al. Zpl: a machine independent programming language for parallel computers. IEEE Trans Software Eng, 2000, 26: 197–211

19 Chamberlain B, Callahan D, Zima H. Parallel programmability and the chapel language. Int J High Perform C, 2007, 21: 291–312

20 Hong C, Chen D, Chen W, et al. Mapcg: writing parallel program portable between cpu and gpu. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques. New York: ACM, 2010. 217–226

21 Chen D, Hong C, Chen W, et al. A mapreduce framework in heterogenous gpu environment. In: Proceedings of the EPHAM09 Workshop. New York: ACM, 2009. 20-27

22 Impact research group. The parboil benchmark suite. http://www.crhc.uiuc.edu/IMPACT/parboil.php