CUDA-NP: Realizing Nested Thread-Level Parallelism in GPGPU Applications

Yi Yang

Department of Computing Systems Architecture
NEC Laboratories America, Inc.
yyang@nec-labs.com

Huiyang Zhou

Department of Electrical and Computer Engineering
North Carolina State University
hzhou@ncsu.edu

Abstract

Parallel programs consist of series of code sections with different thread-level parallelism (TLP). As a result, it is rather common that a thread in a parallel program, such as a GPU kernel in CUDA programs, still contains both sequential code and parallel loops. In order to leverage such parallel loops, the latest Nvidia Kepler architecture introduces dynamic parallelism, which allows a GPU thread to start another GPU kernel, thereby reducing the overhead of launching kernels from a CPU. However, with dynamic parallelism, a parent thread can only communicate with its child threads through global memory and the overhead of launching GPU kernels is non-trivial even within GPUs.

In this paper, we first study a set of GPGPU benchmarks that contain parallel loops, and highlight that these benchmarks do not have a very high loop count or high degrees of TLP. Consequently, the benefits of leveraging such parallel loops using dynamic parallelism are too limited to offset its overhead. We then present our proposed solution to exploit nested parallelism in CUDA, referred to as CUDA-NP. With CUDA-NP, we initially enable a high number of threads when a GPU program starts, and use control flow to activate different numbers of threads for different code sections. We implemented our proposed CUDA-NP framework using a directive-based compiler approach. For a GPU kernel, an application developer only needs to add OpenMP-like pragmas for parallelizable code sections. Then, our CUDA-NP compiler automatically generates the optimized GPU kernels. It supports both the reduction and the scan primitives, explores different ways to distribute parallel loop iterations into threads, and efficiently manages on-chip resource. Our experiments show that for a set of GPGPU benchmarks, which have already been optimized and contain nested parallelism, our proposed CUDA-NP framework further improves the performance by up to 6.69 times and 2.18 times on average.

PPoPP'14, February 15–19, 2014, Orlando, Florida, USA.

http://dx.doi.org/10.1145/2555243.2555254

Categories and Subject Descriptors D.3.3 [**Programming Languages**]: Processors – Compilers, Optimization.

General Terms Performance, Design, Experimentation, Languages.

Keywords GPGPU; nested parallelism; compiler; local memory;

1. Introduction

Parallel programs consist of series of code sections with different thread-level parallelism (TLP). Depending on application characteristics and the parallelization strategy, a parallel thread itself may contain both serial code and parallel loops. Such parallel loops inside a thread are referred to as nested thread-level parallelism. To exploit such nested parallelism in GPGPU (general purpose computation on graphics processing units) applications, the latest Nvidia Kepler architecture introduces the support for dynamic parallelism, which enables a GPU thread to invoke another kernel during execution. Although dynamic parallelism reduces the overhead of invoking a GPU kernel from a CPU, two key limitations remain. First, the communication between a parent thread and its child threads has to be through global memory variables. Second, launching a kernel from a GPU thread involves the device runtime [27] and has a non-trivial performance overhead.

In this paper, we first study a set of benchmarks to show that they contain parallel loops with relatively small loop counts. As a result, the benefits from parallelizing such loops using dynamic parallelism fail to overweigh its overhead. Then, we propose our solution, referred to as CUDA-NP, to exploit nested parallelism within GPGPU applications. Similar to dynamic parallelism, two fundamental

challenges face CUDA-NP: (1) how to have different numbers of threads running in different code sections, and (2) how to enable low-latency data communication between a parent/master thread and its child/slave threads. To address these challenges, CUDA-NP first re-maps threads in a thread block (TB) into a one-dimension organization. Then, for each thread, referred to as a master thread, CUDA-NP adds a set of slave threads along a different dimension. The purpose of the slave threads is to help their master thread on its parallel loops. To do so, CUDA-NP introduces control flow to disable slave threads during sequential code sections. In CUDA-NP, low cost data communication between a master thread and its slave threads is achieved through registers or shared memory. In a way, CUDA-NP can be viewed as lightweight dynamic parallelism.

Our proposed CUDA-NP is implemented as a source-to-source compiler framework, which takes CUDA kernels with OpenMP-like directives as the input and outputs optimized CUDA kernels to exploit nested parallelism. This way, a GPGPU application developer only needs to add pragmas to identify parallel loops within a kernel to take advantage of CUDA-NP.

Our experimental results on Nvidia GTX 680 GPUs show our proposed CUDA-NP achieves remarkable performance gains, up to 6.69 times and 2.18 times on Our optimized code also consistently outperforms the highly optimized library CUBLAS V5.0 on the benchmarks matrix-vector multiplication and transpose-matrix-vector multiplication for different input sizes.

In summary, our work makes the following contributions. (1) We study a set of GPGPU applications and highlight the characteristics of their nested parallelism; (2) we propose simple pragmas and a set of optimization techniques to support nested parallelism; (3) we implement our CUDA-NP using a source-to-source compiler to relieve the programming complexity from application developers; and (4) we show that our proposed solution is highly effective and significantly improves the performance.

The remainder of the paper is organized as follows. In Section 2, we present a brief background on GPGPU architecture with a focus on Nvidia dynamic parallelism. We also analyze a set of GPGPU application to show the characteristics of their parallel loops. In Section 3, we present our compiler framework to exploit nested parallelism. The experimental methodology is addressed in the Section 4 and the results are presented in Section 5. Related works are discussed in Section 6. Section 7 concludes our paper.

2. Background

2.1 GPGPU Architecture and Programming Model

In order to achieve high computational throughput and memory bandwidth, GPGPU exploits many-core architectures and organizes the cores in a two-level hierarchy. First, a GPU contains multiple streaming multiprocessors (SMs) in Nvidia GPU architecture. An SM is also called a next generation SM (SMX) in Nvidia's latest Kepler architecture and is similar to a compute unit (CU) in AMD GPU architecture. Each SMX/CU in turn consists of multiple streaming processors (SPs) or thread processors (TPs). An SMX/CU can support thousands of threads running concurrently following the single-program multiple-data (SPMD) programming model.

In the CUDA programming model, the threads are also managed in a two-level hierarchy: thread grids and thread blocks (TBs). A GPGPU program, also called a kernel, is launched as a grid of TBs. A TB in turn contains multiple threads, which can have up-to-three-dimension thread identifiers (ids). In our compiler, we always map a multi-dimension thread id into a one-dimension one using the approach presented in Section 3.7. Therefore, in our subsequent discussions, we assume the input kernel has only one-dimensional threads in a TB.

GPGPU employs the single-instruction multiple-data (SIMD) model to amortize the cost of instruction decode and fetch. A small group of threads, referred to as a warp, share the same instruction pointer. The latest Nvidia Kepler architecture introduces a set of shuffle (shfl) instructions to enable the data exchange through registers for threads in the same warp. One shfl instruction used in this paper is __shfl(var, laneID, laneSize). For this instruction, a warp (32 threads) is partitioned into small groups with the group size as laneSize. Then, the laneID is used to specify the relative thread id in a group, and the var is the variable to be read. For example, the instruction __shfl(var, 0, 4) means that a warp contains 8 groups with a group size of 4 and all threads in the same group will read var from the first thread of the group. As a result, threads with id 0, 1, 2, and 3, belonging to the first group, read 'var' from thread 0; threads with id 4, 5, 6, and 7, read 'var' from thread 4; and so on. Compared to shared memory, which can be shared among all threads in the same TB, the shfl instructions have higher performance with the following two limitations. First, it can only be supported for the threads in the same warp. Second, threads can read a register from another thread in the same warp, but cannot write to a register of another thread.

The support for dynamic parallelism is introduced to Nvidia GPUs with compute capability 3.5. With dynamic parallelism, a GPU thread can launch a kernel during execution. Dynamic parallelism provides an easy way to develop GPU kernels for a program that contains nested parallelism without involving the host CPU. However, in order to achieve the high performance, the kernel launched by a GPU thread must have a very high number of threads to offset the overhead of launching a kernel. To illustrate the overhead of dynamic parallelism, we use the memory-copy

micro-benchmark in our experiment on an Nvidia Tesla K20c GPU. To copy 64-million floats, the baseline micro-benchmark without dynamic parallelism achieves the bandwidth is 142 GB/s. Then, we observe that once we enable the compiler flag for dynamic parallelism, the original kernel without using dynamic parallelism can only achieve 63 GB/s. Such overhead is referred to as dynamic-parallelism-enabled kernel overhead [27]. Next, we modify the benchmark to make use of dynamic parallelism. In the dynamic parallelism version, we have a parent kernel and a child kernel. The parent kernel is launched once, but each thread of parent kernel will launch a child kernel. the child kernel can be launched many times. In the child kernel, each thread simply copies a float from the input to the output. If the number of threads of the parent kernel is m, and the number of threads of every child kernel launch is n, then m*n is the overall floats to be copied from the input to the output. We fix the value of m*n to 64 million and show the bandwidths for different m in Figure 1. Although the overall workload remains the same, the performance degrades rapidly when the number of child kernel launches increases. In other words, each kernel launch needs to have a high number of threads to achieve good performance. From Figure 1, we can see that when each child kernel launch has 16k threads, the overall memory copy bandwidth only reaches 34GB/s. This highlights the kernel launching overhead for dynamic-parallelism. Another limitation of dynamic parallelism is that the communication between the parent thread and its child threads has to be through global memory [27].

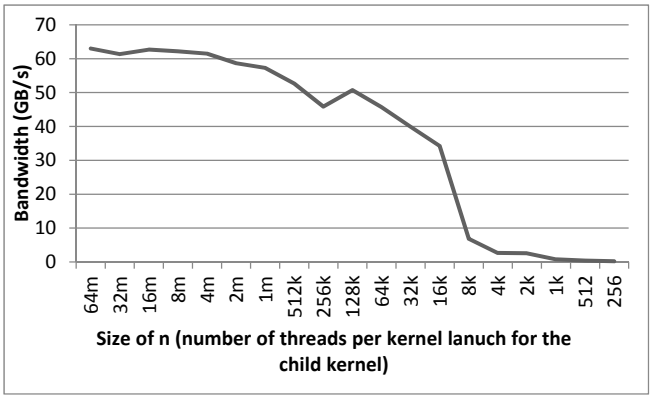

Figure 1. The throughput of the memory-copy micro-benchmark using dynamic parallelism.

2.2 Nested Parallelism in GPGPU Programs

GPGPU applications are typically highly parallelized due to the required TLP to hide high memory access latencies. Still, there exist parallel loops in the kernel code. As an example, Figure 2 shows the kernel code of transposed-matrix-vector multiplication (TMV). Each thread computes one element in the output vector. The loop between lines 4 and 5 reads one column of input matrix *a* and the vector *b*, and performs the dot-product operation. This example illustrates common reasons for nested parallelism existed in GPU kernels.

```
1  __global__ void tmv(float *a, float*b, float* c, int w, int h){
2    float sum = 0;
3    int tx = threadIdx.x+blockIdx.x*blockDim.x;
4    for (int i=0; i<h; i++)
5      sum += a[i*w+tx]*b[i];
6    c[tx] = sum;
7  }
```

Figure 2. The kernel code of transposed-matrix-vector multiplication (TMV).

First, developers tend to view that there is already sufficient TLP. In the TMV example, the output vector c may have a high number of elements and further parallelization of the loop in Figure 2 may be considered not necessary. However, based on the characteristics of the applications, each thread may require too much resource to limit the number of threads that can run concurrently on each SMX, even though the overall number of threads of the application is indeed high enough.

Second, if a loop contains loop-carried dependencies, application developers may choose not to parallelize it. As shown in Figure 2, to parallelize the loop, the reduction primitive needs to be supported.

Third, if an application developer chooses to further parallelize some parallel loops in a kernel, he/she needs to understand the GPGPU architecture very well, in order to achieve good performance. For example, if a parallel loop is distributed among threads in the same warp, we should avoid workload imbalance due to the nature of SIMD execution. Another key factor is how to balance the resource usage among shared memory, register file and local memory. Therefore, in this paper, we argue for a compiler approach to exploit nested parallelism to relieve the application developers from the associated complexity.

Overall, the loop in Figure 2 showcases a common example in many benchmarks that contain parallel loops. An outstanding feature of a good candidate for leveraging nested parallelism is limited TLP due to the nature of the application or the heavy resource usage of each GPU

2.3 Characteristics of Nested Parallelism in GPGPU Programs

In order to understand available nested parallelism in GPGPU programs, we studied the benchmarks in Nvidia SDK [26], Rodinia [6] and GPGPUSim [2]. The detailed experimental methodology is discussed in Section 5. We list in Table 1 some benchmarks that contain nested parallelism. In the table, the first column (**Name**) contains the name of the benchmarks; the second column (**Input)** shows the input to each benchmark; the third column (**PL**) shows how many parallel loops exist in the kernel; the forth col-

umn (**LC**) shows the largest loop counts among these parallel loops; the fifth column (**R/S**) indicates if the loops contain the scan(S)/reduction(R) operations or not (X). We also show the resource usage including register file (**REG**), shared memory (**SM**) and local memory (**LM**) per thread for the baseline code (**BL**) and optimized versions (**OPT**) generated from CUDA-NP.

From Table 1, we can see that the benchmarks LE, LIB and CFD have intensive local memory usage to limit their performance, even though they do have relatively high numbers of concurrent threads running on each SMX; the benchmarks LU, MV, SS and BK have intensive shared memory usage, which limits the number of concurrent threads on each SMX; the benchmark MC has intensive usage of both shared memory and local memory; the remaining benchmarks, TMV and NN, do not have intensive resource usage. From these benchmarks, we can see that the loop counts of parallel loops are relatively small.

Table 1. Benchmarks

Name	Input	PL	LC	R/S	Bytes per thread (BL)			Bytes per thread(OPT)		
					REG	SM	LM	REG	SM	LM
MC	grid=8	4	12	X	252	288	40	144	36	0
LU	2048.dat	4	32	R	44	96	0	72	24	0
LE	testfile.avi	3	150	R	156	0	600	252	4	24
MV	2K*2K	1	32	R	100	132	0	100	34	0
SS	DIM=8K	2	8K	R	60	80	0	72	20	0
LIB	NPATH=256K	4	80	S	216	0	960	200	40	640
CFD	fvcorr.domn.193K	1	4	R	252	0	56	252	0	8
BK	2M	2	32	X	60	128	0	56	4	0
TMV	2K*2K	1	2K	R	88	0	0	64	4	0
NN	1K	1	1K	R	88	0	0	56	0	0

3. CUDA-NP: A Directive-Based Compiler Framework for Nested Parallelism

In this section, we present our CUDA-NP compiler framework to leverage nested parallelism. The input to our compiler is a CUDA kernel with OpenMP-like directives to denote parallel loops. The output of our compiler framework is the optimized kernel code. Figure 3 shows an example. The kernel with directives denoting the parallel loop is shown in Figure 3a and the optimized kernel using our proposed compiler is shown in Figure 3b. The first transformation on the input kernel is to increase the TB size. Given the hardware limit of GTX 680 GPUs, the maximal number of threads in a TB is 1024. So, the TB size is increased up to 1024 if the input kernel has a smaller TB size. As our compiler already re-maps the threads in a TB into a one-dimension organization, increasing the TB size is achieved by adding a new dimension. The newly introduced threads will be used to carry out the parallel loops in the input kernel. To differentiate the original threads from the newly added ones, we refer to the original threads as '*master*' threads and the newly added ones as '*slave*' threads and we use '*master_id*' and '*slave_id*' as their thread ids along different dimensions. If the master threads are aligned in the X/Y dimension, slave threads will be added along the Y/X dimension, as discussed in Section 3.4. For the kernel shown in Figure 3, master_id is actually threadIdx.x and slave_id is threadIdx.y.

```
1  #define BLOCK_SIZE 16
2  __global__ void lud_perimeter(float *m, int matrix_dim, int
3  offset) {
4     __shared__ float
5     peri_row[BLOCK_SIZE][BLOCK_SIZE],
6     peri_col[BLOCK_SIZE][BLOCK_SIZE],
7     dia[BLOCK_SIZE][BLOCK_SIZE];
8     …….
9     int array_offset;
10    array_offset = offset*matrix_dim+offset;
11
12    // parallel loop
13    #pragma np parallel for
14    for (i=0; i < BLOCK_SIZE; i++)
15     peri_row[i][idx]= m[array_offset+
16        (blockIdx.x+1)*BLOCK_SIZE+matrix_dim*i];
17    ……
18 }
```

(a) The input kernel

```
1  #define BLOCK_SIZE 16
2  __global__ void lud_perimeter_np(float *m, int matrix_dim,
3  int offset) {
4     __shared__ float
5     peri_row[BLOCK_SIZE][BLOCK_SIZE],
6     peri_col[BLOCK_SIZE][BLOCK_SIZE],
7     dia[BLOCK_SIZE][BLOCK_SIZE];
8     …….
9     int array_offset;
10    if (slave_id==0) { //slave_id == 0 means a master thread
11       array_offset = offset*matrix_dim+offset;
12    }
13    array_offset = read_from_master(array_offset);
14    for (i=slave_id; i < BLOCK_SIZE; i+=slave_size)
15     peri_row[i][idx]= m[array_offset+
16        (blockIdx.x+1)*BLOCK_SIZE+matrix_dim*i];
17    ……
18 }
```

(b) The output kernel from CUDA-NP

Figure 3. The kernel 'lud_perimeter' before and after CUDA-NP optimizations.

The 'lud_perimeter' kernel in Figure 3a has only 32 threads in a TB and each TB needs 3kB shared memory. As a result, 16 thread blocks can run concurrently on each SMX on an Nvidia GTX 680 GPU, for which the shared memory is configured to 48kB per SMX. The line 13 in Figure 3a is the pragma indicating the subsequent loop can be parallelized. Then in the code optimized using CUDA-NP, as shown in Figure 3b, each TB has 32x8 threads, where the size of the *X* dimension is 32 and the size of the *Y* dimension is 8. However, we only allow 32 threads in a TB to be active for the sequential code such as line 10 in

Figure 3a in the kernel. For the parallel loops, such as those between line 14 and 16 in Figure 3a, all 256 threads per TB are active, and each thread processes one or more iterations of the loop. In other words, for each master thread, 7 slave threads are added to share its parallel workload. Note that the parameter '*slave_size*' is 8 as 8 threads (1 master + 7 slaves) will all work equally on the parallel loops.

Next, we use one example to illustrate why our optimized kernel improves performance, as shown in Figure 4. In Figure 4a, we assume each TB of the input kernel has 8 threads running through its whole lifetime. Then in the optimized kernel, each TB has 8*4 threads as shown in Figure 4b. However we have only 8 threads running for sequential codes, such as line 10 in Figure 3a, and 8*4 threads for loop sections, such as line 14 to line 16 in Figure 3a. These active threads that execute the sequential sections are the master threads, as they are very similar to original threads in the input kernel. But in the parallel loop sections, the workload of each thread in the input kernel will be distributed to the corresponding master thread and its slave as shown in the Figure 4b. The overall performance is improved due to higher TLP for parallel loops.

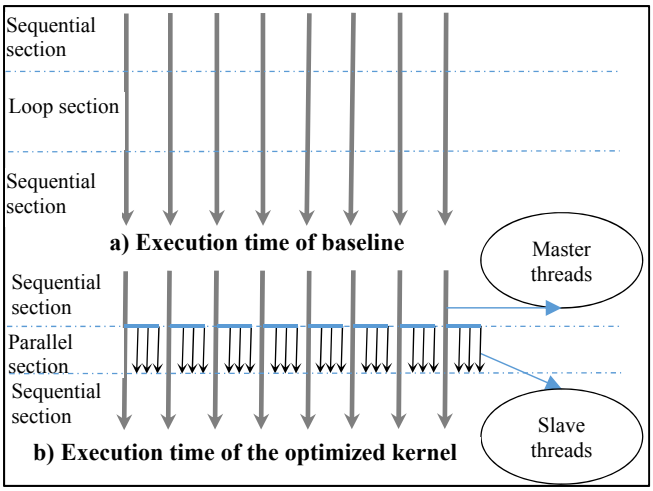

Figure 4. Execution paradigms to show why CUDA-NP works.

In the example in Figure 3, the master threads are aligned along the *X* dimension, i.e., the master id is threadIdx.x in the corresponding TB of the input kernel. When we generate the slave threads for a master thread, they share the same master id (i.e., threadIdx.x) but have different slave ids (i.e., threadIdx.y). Therefore, we actually use threads in different warps to work collaboratively for the workload of a master thread in the input kernel, since the warps are formed using consecutive thread ids. For example, the threads with two-dimension ids (1,0) to (1,7) in a TB of our optimized kernel perform the workload of the thread with id 1 in the TB of the original kernel. The thread with id (1,0) in the optimized kernel will be the master thread corresponding to thread 1 in the original kernel and threads with ids (1,1) to (1,7) are its slave threads. These threads will be in different warps as the TB dimension is 32x8. Therefore, we refer to this way of distributing parallel loop iterations as **inter-warp NP.** Besides inter-warp NP, we can map the 'master_id' to threadIdx.y and map 'slave_id' to threadIdx.x. For example, we use threads with thread ids (0,1) to (7,1) in our optimized kernel to perform the workload of thread 1 in the original kernel. This way, we use threads within a warp to distribute the parallel loops. We refer to this way of thread id mapping or workload distribution as **intra-warp NP**.

As illustrated in Figure 3b, several key code transformations are performed by our proposed CUDA-NP compiler framework. In the sequential code sections, it introduces the control flow in line 10 to only allow the master threads to compute the variable '*array_offset'*. Since this variable is to be used in the parallel loop by slave threads, it needs to be broadcasted to the slave threads. A function called *read_from_master* is introduced for this purpose. In the parallel loop, it updates the loop iterator using slave ids and the number of slave threads (i.e., '*slave_size*') so that multiple slave threads can process multiple loop iterations in parallel. As the loop bound checking remains in the transformed code, this transformation is valid if the loop bound is determined at the runtime. Also, from this example, we can see that a key challenge of our code transformations is to handle the variables which are live cross a parallel section and a sequential section.

In Section 3.1, we discuss how our compiler handles scalar live-in variables. Section 3.2 addresses the scalar live-out variables. For live array-variables, if they reside in global memory or shared memory, they are already accessible by the master threads and their salve threads. So, in Section 3.3, we discuss our compiler transformation to deal with live array-variables, which are located in local memory. In Section 3.4, we summarize the tradeoffs between the two workload distribution schemes, intra-warp NP and inter-warp NP. In Section 3.5, we present the overall compiler algorithm for code transformations. Section 3.6 lists our proposed pragmas for NP. Section 3.7 discusses the preprocessing step of our compiler.

3.1 Scalar Inputs/Live-Ins to Parallel Sections

For a scalar variable defined in a sequential section and used in a subsequent parallel section, i.e., a live-in variable to the parallel section, we need to broadcast it from a master thread to its slave threads. The exception is that the variable is in the global memory which is already visible to all the threads. For example, as shown with line 15 in Figure 3a, the global memory array *m* can be directly accessed by slave threads. Shared memory has similar behavior and

can be used directly by slave threads without additional code transformations.

The variables in the register file or local memory have to be broadcasted to slave threads as they are private to a master thread. We implement it using the function *read_from_master*. If we use intra-warp NP for Kepler GPUs, since the master and its slave threads are in the same warp, we can use the shfl instruction, *__shfl(var, 0, slave_size)*, to implement the *read_from_master* function. As explained in Section 2.1, for such a __shfl instruction, a warp threads first are partitioned into small groups with the group size of *slave_size*. Therefore, a group actually contains all the slave threads of a master thread for intra-warp NP. Then all threads within a group will read the value of *var* from the master thread, whose id (i.e., threadIdx.x % slave_size) is 0, in the small group. For inter-warp NP or intra-warp NP on GPUs that do not support __shfl instructions, *read_from_master* is implemented using shared memory. In this case, master threads first write values to shared memory, and then slave threads read from shared memory.

Instead of communicating through registers or shared memory, another way is to let all slave threads compute the live-in variables redundantly. Such redundant computation is also called uniform vector operations as they have the same input and output values for different threads [7]. If the overhead is only simple ALU computations like line 10 in Figure 3a, in general redundant computation can deliver better performance due to eliminating the shared memory usage and control flow. In our compiler, if an instruction's inputs are constant values or the output of a uniform vector instruction, this instruction will be executed by all slave threads redundantly. Otherwise, we let the master thread execute it and broadcast the result to slave threads.

3.2 Scalar Outputs/Live-Outs of Parallel Sections

Similarly to the live-ins of a parallel section, if a scalar output of a parallel section is in global memory or shared memory, we can just leave as it is, as it is already visible to the master threads. If a variable is in the register file or local memory, we have different scenarios to handle. One common scenario is a reduction or scan variable, for which we can generate the parallel implementations to retrieve the results from slave threads. We implement the reduction using shared memory for inter-warp NP or using the shfl instruction for intra-warp NP. For the scan implementation, we also use a similar approach to Nvidia CUDA SDK [26].

There are scenarios that an output of a parallel section is neither a reduction variable nor a scan variable. One such example is the code '*if (i==3) x = a[i];*' inside a parallel loop, where *i* is the loop iterator and the variable *x* is the used in later sequential sections. The problem with this code is that in our CUDA-NP scheme, each slave thread has a local variable *x* and will execute the code. But it is supposed that only one slave thread will write the value to *x*, and the *x* of this slave thread needs to be transferred to the master thread. In such a case, we can make the initial value of *x* to be 0 so that a reduction operation on *x* can be used to retrieve the value from the slave threads.

3.3 Live Array-Variables Residing in Local Memory

Since the register file has a limited size and cannot be accessed as an indexed array, array variables residing in the local memory are used in some CUDA programs. As shown in Figure 5, the array *Grad* has to be spilled into local memory due to the register file size limitation. Such local memory accesses incur high pressure on the L1 cache and lead to poor performance.

```
1  #define NPOINTS 150
2  __global__void ellipsematching_kernel(…) {
3      float Grad[NPOINTS];  //live array-variable in local memory
4      ……
5      #pragma np parallel for
6      for(n = 0; n < NPOINTS; n++) {
7          ……
8          Grad[n] = tex1Dfetch(t_grad_x,addr) * ……;
9      }
10     #pragma np parallel for reduction(+:sum)
11     for(n = 0; n < NPOINTS; n++) sum += Grad[n];
12     ave = sum / ((float) NPOINTS);
13
14     #pragma np parallel for reduction(+:var,ep)
15     for(n = 0; n < NPOINTS; n++) {
16         sum = Grad[n] - ave;
17         var += sum * sum;
18         ep += sum;
19     }
20     ……
21     if(((ave * ave) / var) > sGicov)
22         gicov[(i * grad_m) + j] = ave / sqrt(var);
23 }
```

Figure 5. The kernel with live array-variables in local memory.

We apply our CUDA-NP on the parallel loops marked with our CUDA-NP pragmas in Figure 5, and Figure 6 shows the code after our optimization. From Figure 6, we can see all parallel loops are distributed to multiple slave threads. For the loop starting from line 6 in Figure 5, each slave thread only needs to compute *NPOINTS/slave_size* iterations as shown from line 6 in Figure 6. As shown from line 7 in Figure 6, each iteration of the loop in a slave thread is mapped to an iteration of the loop of the baseline kernel before our optimization. This way, all iterations of the loop in the baseline are distributed to slave threads. The reduction or scan operations are also appended after the loops if the pragmas specify the reduction or scan clauses.

As we discussed in section 3.1, a local array is private to a thread, and not visible to other threads. However, in order for slave threads to process a parallel loop, this array has to be shared among those threads. Therefore, we need to re-

place a local array with a shared memory array or a global memory so as to make it visible to all threads. One exception is that a local array is accessed based on the loop iterator. For example, the parallel loops in Figure 5 always access the array *Grad* using the loop iterators. In this case, since each slave thread only needs to accesses part of the local array without interleaving, we can partition the local array into small ones and distribute each small array to one slave thread. Therefore, for a live local array, we can replace it with a global memory array, a shared memory array, or partition it into small local arrays as shown in Figure 6. Since these approaches only affect the accesses to local arrays, we differentiate them using two MACROs: *DEF_Grad* and *Grad(i),* in Figure 6a, 6b, and 6c so that the code in Figure 6d remains the same.

1. ***Replace a local array with a global memory array***: We first define a new global array and partition it such that each partition corresponds to the local array of a master thread. As shown in Figure 6a, the MACRO *DEF_Grad* partitions a new global memory array *Grad_g* based on the id of a master thread so that all slave threads of the master thread access the same partition.
2. ***Replace a local memory with a shared memory array***: In this case, we first declare a shared memory array. The size of its first dimension is the *master_size*, i.e., the number of master threads in a TB, and the size of its second dimension is the size of the local array. Then slave threads can access the shared memory based on its master thread id and the index of original local array. Since many benchmarks already use shared memory intensively, the potential issue of this approach is the increased usage of shared memory.
3. ***Partition a local array into smaller local arrays***: In Figure 6c, each slave thread only requires a smaller local array whose size is *NPOINTS/slave_size*. This approach requires a slave thread must only read and write its own local array after the partition.

Our framework employs the following policy to decide which option is to be used to replace a local array. First, if the local array meets all conditions to be partitioned into smaller ones, we choose option 3. Otherwise, the size of the local array is checked, and the shared memory is used to replace the local array, if the size of the local array is less than 384 byte. The reason for this choice is that assuming the local array size is 384 bytes and we can launch 8 slave threads for each master thread, 48Kbytes shared memory can support 128 master threads and 896 slave threads after our optimizations, which provides enough TLP on each SMX. If the shared memory is already used in the baseline, we also need to subtract such shared memory usage from 384 bytes to ensure that shared memory will not be the resource bottleneck for TLP. The last choice is to replace the local array with one in global memory due to the high access latency.

```
#define DEF_Grad  float* Grad=Grad_g+ \
   (master_size*blockIdx.x)* NPOINTS+master_id
#define Grad(i) Grad[i*master_size]
```

(a) Replace a local memory array with a global memory one

```
#define DEF_Grad  __shared__ float Grad[master_size][ NPOINTS]
#define Grad(i) Grad_sm[master_id][i]
```

(b) Replace a local memory array with a shared memory one

```
#define DEF_Grad float Grad_reg[NPOINTS/slave_size]
#define Grad(i) Grad_reg[i%(slave_size)]
```

(c) Partition a local array to small ones

```
 1  #define NPOINTS 150
 2  template<int slave_size>
 3  __global__ void ellipsematching_kernel(…, float*Grad_g) {
 4      DEF_Grad
 5      ……
 6      for(ni = 0; ni < NPOINTS/slave_size; ni++) {
 7          n = ni*slave_size+slave_id;//map thread id to iteration
 8          …...
 9          Grad(n) = tex1Dfetch(t_grad_x,addr) * ……;
10      }
11      for(ni = 0; ni < NPOINTS/slave_size; ni++) {
12          n = ni*slave_size+ slave_id;
13          sum += Grad(n);
14      }
15      sum =reduction(sum);// reduction on slave threads
16      ave = sum / ((float) NPOINTS);
17
18      for(ni = 0; ni < NPOINTS/slave_size; ni++)  {
19          n = ni*slave_size+ slave_id;
20          sum = Grad(n) - ave;
21          var += sum * sum;
22          ep += sum;
23      }
24      var =reduction(var);//
25      ep = reduction(ep);//
26      ……
27      if (slave_id==0)  // only master threads
28       if(((ave * ave) / var) > sGicov)
29          gicov[(i * grad_m) + j] = ave / sqrt(var);
30  }
```

(d) Optimized code

Figure 6. Approaches to handle live array-variables in local memory.

3.4 Inter-Warp NP vs. Intra-Warp NP

The choice between inter-warp NP and intra-warp NP may have significant performance impact. Here, we summarize their tradeoffs. First, since threads in the same warp can use registers to exchange data, __shfl instructions can be used for communication and also the scan and reduction operations for intra-warp NP. As a result, the intra-warp NP may have less shared memory usage. Second, if the slave threads of a master thread have different workloads, the intra-warp NP will be worse than inter-warp NP due to control divergence. Third, intra-warp NP may have negative impact on memory coalescing as it changes the memory access pattern of the original kernel. In general, the master threads in the original kernel have adjacent

thread ids and tend to access the global memory in a coalesced way. If we map these master thread ids into threadIdx.y as the intra-warp NP approach, these coalesced global memory accesses are broken. Forth, a similar issue may also happen for constant memory accesses when we use intra-warp NP. Considering line 11 in Figure 5, if the *Grad* is a constant array, threads in a warp will access the same address of *Grad* in the baseline. However, after intra-warp NP, slave threads of a master thread will access different addresses of the constant array. Such accesses cannot leverage the hardware broadcast logic and may hurt performance. Finally, to use the __shfl instructions, the number of slave threads for a master thread has to be (a 2's power -1), i.e., 1, 3, 7, 15. Otherwise, these slave threads might be in different warps.

```
NP_transformation(Kernel kernel)
  css = generateCodeSections(kernel)
  inter-warp or intra-warp thread map for kernel (Section 3.4)
  for cs in css:
    if cs is sequential:
      cs is master thread model
    if cs is a parallel loop:
      map each slave thread id to iterations of cs
      for each input in of cs:
        insert broadcast function for in before cs (Section 3.1)
      for each output out of cs:
        insert reduction or scan for out after cs (Section 3.2)
      for each live local memory array lm : (Section 3.3)
        map lm to global memory, shared memory or the
        register file
```

Figure 7. The overall compiler algorithm of CUDA-NP.

3.5 Compiler Algorithm

Here, we summarize our CUDA-NP compiler algorithm, as shown in Figure 7. CUDA-NP takes a kernel as the input. It parses the kernel into a series of code sections. Each code section is either sequential or parallel. A parallel section is identified by the '*np*' pragma. First, we map the thread id of the input kernel to master and slave thread ids in the transformed kernel for either the inter-warp NP or intra-warp NP approach. Then, if a code section is sequential one, we generate the control flow to only allow the master threads to execute it. Redundant computations can be used in sequential sections depending on the characteristics of an instruction as discussed in Section 3.1. For parallel sections, all slave threads along with their master threads are active. For each parallel section, we also generate the code for its scalar input (Section 3.1) and the code for its scalar output (Section 3.2). The live local arrays have to be replaced with global/shared memory arrays, or partitioned into smaller local arrays, as discussed in Section 3.3.

3.6 Pragma

In order to reduce the programming complexity to leverage nested parallelism, we adapt the OpenMP pragmas for our CUDA-NP framework. Most of CUDA-NP grammars are designed to be very similar to OpenMP pragmas on purpose. A developer can add '*#pragma np for*' to denote a parallel loop, and can also specify different clauses of the pragma. A *copy-in* clause defines the data which should be broadcasted from a master thread to its slave threads. If a *copy-in* clause is not available from users' pragmas, our compiler can automatically find the live-in variables defined before a parallel loop and make them to be broadcasted from a master thread to its slave threads. A *reduction/scan* clause defines the reduction or scan operations. Developers have the flexibility to specify the preferred number of slave threads (*number_threads*), whether the inter-warp NP or intra-warp NP is preferred (*NP_type*), and the targeted version of Nvidia CUDA compute capability (*sm_version*). Our current support for compute capability versions is mainly for the purpose of using shfl instructions. If the target version is less than 3, the shfl instruction cannot be used to guarantee correctness. If a developer does not provide such information, our compiler generates multiple versions to explore different numbers of slave threads, and different thread distribution approaches.

```
threadIdx_x ← threadIdx.z * blockDim.x * blockDim.y +
threadIdx.y * blockDim.x + threadIdx.x
(a)  map three-dimension thread ids into one-dimension ones
threadIdx_x ← threadIdx.x % blockDim_x
threadIdx_y← (threadIdx.x/blockDim_x) % blockDim_y
threadIdx_z←threadIdx.x /(blockDim_x * blockDim_z)
(b)  map one-dimension thread ids into three-dimension ones
```

Figure 8. Mapping thread ids.

```
vertexInterp2(isoValue, v[0], v[1], ...);
vertexInterp2(isoValue, v[1], v[2], ...);
vertexInterp2(isoValue, v[2], v[3], ...),
……
vertexInterp2(isoValue, v[3], v[7], ...),
                    (a) Sequential code
__constant__ int CS_0= {0,1,2,…,3}
__constant__ int CS_1= {1,2,3,…,7}
for (int i=0; i<12; i++)
vertexInterp2(isoValue, v[CS_0[i]], v[CS_1[i]], ...);
          (b) A loop converted from the code in (a)
```

Figure 9. Converting sequential code into a loop.

3.7 Preprocessors

The purpose of the preprocessors to our compiler is to generate the input source code suitable for our code optimizations.

1. ***Convert a TB with multi-dimensional threads into a TB with one-dimensional threads***: We use the mapping relationship shown in Figure 8 to map multi-dimension thread ids to one-dimension ones and vice versa. This transformation has limited performance impact since it does not change thread organizations within warps. In other words, the threads in a warp remain in a warp after

the transformation. Therefore, it does not affect memory coalescing or divergence.

2. ***Combine unrolled statements into a loop***: We found that sometimes the developers may manually unroll some loops. Since our compiler targets at parallel loops, for statements after unrolling, they can be combined into a parallel loop to take advantage of CUDA-NP. Figure 9a shows such an example, as the input of each statement cannot be mapped to an iterator of a loop directly. In our pre-processor, we put the non-linear indexes in constant buffers, and then access these indexes using loop iterator. This way, we can convert such sequential code into a parallel loop.
3. ***Pad arrays***: as shown in Figure 5, the size of the local memory array '*Grad*' is 150, which is not multiple of 4,8,16, or 32. However, if we apply the inter-warp NP scheme, the number of slave threads of a master thread has to be (a 2's power number – 1) and the loop count needs to be a multiple of slave_size. In this case, we can pad the size of Grad to 160 and also increase the upper bound of the loop to 160 so that the loop counter is the multiple of 32. Then an additional control flow "*if (i<150)*' is added in the loop body to skip the padding data, where *i* is the loop iterator. Such padding may introduce workload imbalance among slave threads due to some idle iterations.

4. Experimental Methodology

To evaluate our proposed compiler, we perform our experiments on Nvidia GTX 680 GPUs with CUDA SDK 5.0. We let the CUDA runtime determine the shared memory usage automatically based on the resource requirement of each benchmark. Most benchmarks used in the experiments are from Nvidia SDK, GPGPUSim, and Rodinia benchmark suite. Among these benchmarks, MarchingCubes (MC) is from Nvidia SDK, and Libor (LIB) is from GPGPUSim. Lud (LU), Leukocyte (LE), Streamcluster (SS), Computational Fluid Dynamics (CFD), BucketSort (BK), and Nearest Neighbor (NN) are from Rodinia. The LE is the array order version [4], and BK is in the Hybrid Sort package. Since NN only uses one thread in each TB and has very poor performance, we first modify the TB configuration so that each TB has 32 threads, which is 2.89 times faster than the original version. Then we use this modified version as the baseline in our experiments. We use the optimized matrix-vector multiplication (MV) based on [42]. The TMV code is shown in Figure 1. In Table 1, as a comparison, we show the resource usage of our optimized benchmarks. For these benchmarks, we manually add the NP pragma to identify parallel loops.

We implement our proposed CUDA-NP in a source-to-source compiler using Cetus [20]. Our compiler has an auto-tuning mechanism to select from multiple choices, such as intra-warp NP or inter-warp NP, and different numbers of slave threads to be used to distribute parallel loop iterations.

5. Experimental Results

In Figure 10, we report the speedups of the optimized kernel generated by our compiler over the baseline. As shown in the figure, our proposed CUDA-NP can achieve from 1.36 to 6.69 times speedups. On average using the geometric mean (GM), our proposed CUDA-NP can achieve 2.18 times speedup among the ten benchmarks.

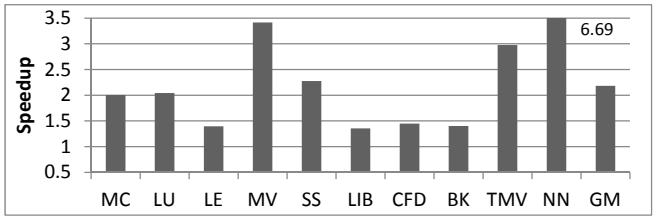

Figure 10. Speedups of our proposed CUDA-NP over baseline.

In order to better understand the impact of our CUDA-NP on these benchmarks, we show results for different slave_sizes coupled with either inter-warp NP or intra-warp NP in Figure 11. Among these benchmarks, LU and NN are the only cases that intra-warp NP achieves better performance than inter-warp NP. The main reason for LU is that the loops of LU are in the control flow 'master_id<16'. The intra-warp NP approach allocates slave threads in the same warp for a master thread. Assuming each master thread has 3 slave threads, each warp will contain only 8 master threads (8 master threads+24 slaves). These 8 master threads and their slaves will execute the same path. Therefore, control divergence disappears after intra-warp NP. Furthermore, when three slave threads are allocated to each master thread (i.e, slave_size=4), it achieves the best performance for both intra-warp NP and inter-warp NP, as it enables 2k threads per SMX based on the resource usage. For NN, the intra-warp NP version can access the global memory in a more coalesced manner while the impact of inter-warp NP is minor.

For other benchmarks, inter-warp NP always outperforms intra-warp NP. MC, LIB and LE have imbalanced workload among slave threads for intra-warp NP. For example, since the loop count of MC is 12, and if we allocate 7 slave threads for each master thread, then some slave threads have to take 2 iterations and others take 1 iteration. LIB and LE have similar behaviour to MC.

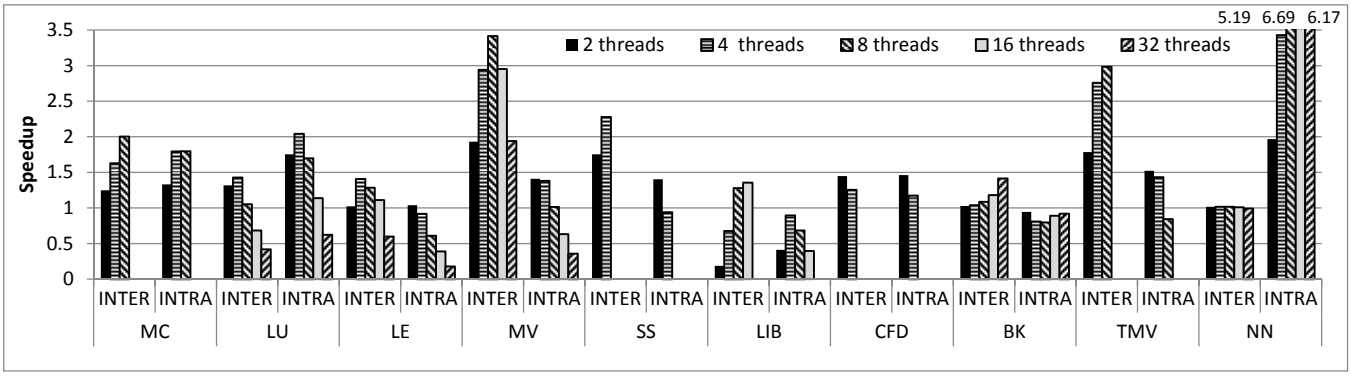

Figure 11. Performance comparison between inter-warp and intra-warp NP with different slave_sizes. Some slave_sizes are not applicable as the resulting TB size would exceed the maximal TB size.

The difference between inter-warp NP and intra-warp NP for CFD is minor, because the memory accesses of the baseline are not all coalesced and the loop iterations can be evenly distributed to slave threads.

From Figure 11, we can also see that in some cases, the performance degrades when the number of slave threads increases. This observation is consistent with the previous work [18] in that higher TLP is not always helpful.

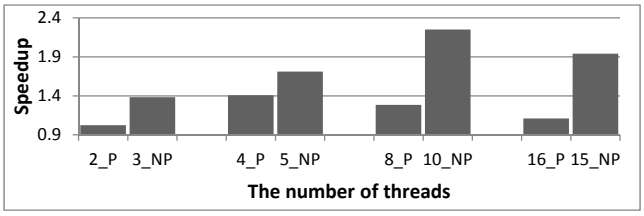

Figure 12. The impact of padding on LE.

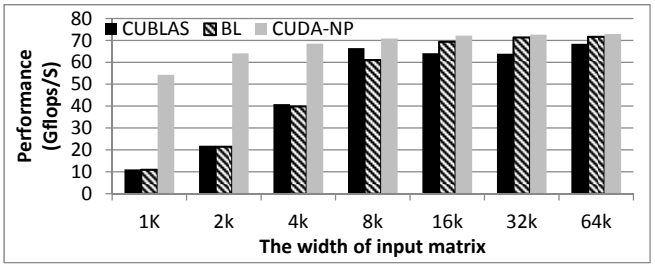

Figure 13. **TMV results on GTX 680 for matrices with variable widths and a constant height (2k).**

As we discussed previously, if the loop count of a parallel loop counter is not power of 2, we need to pad it to the multiple of power of 2 for intra-warp NP. However, for inter-warp NP, such limitation can be removed. Therefore in Figure 12, for the benchmark LE, we compare the results without padding to those with padding. The loop count of the baseline is 150. We choose to compare these results that have similar numbers of slave threads. If we compare 3 threads (no padding) to 2 threads (padding), 5 threads (no padding) to 4 threads (padding), 8 threads (padding) to 10 threads (no padding), 16 threads (padding) to 15 threads (no padding), we can see that the no-padding version (**NP**) always outperforms the padding version (**P**). The best optimized version can achieve 2.25 times speedup over the baseline.

Since the benchmarks TMV and MV are available in Nvidia CUBLAS library, we also compare our optimized version to CUBLAS V5.0 [25] in Figure 13 and Figure 14 on GTX 680 GPUs. We also include the SMM version for MV [42] in Figure 14 as a reference. In Figure 13, the height of input matrix is always 2k, and we vary the width of the input matrix. From this figure we can see that our baseline has similar performance to CUBLAS, and our CUDA-NP solution achieves significantly better performance. For matrices with smaller sizes, since the number of overall threads is determined by the width of input matrix, our approach enables more threads to occupy the SMXs and achieve even higher speedups. For example, if the input width is 1k, our CUDA-NP version delivers 4.9 times speedup over CUBLAS.

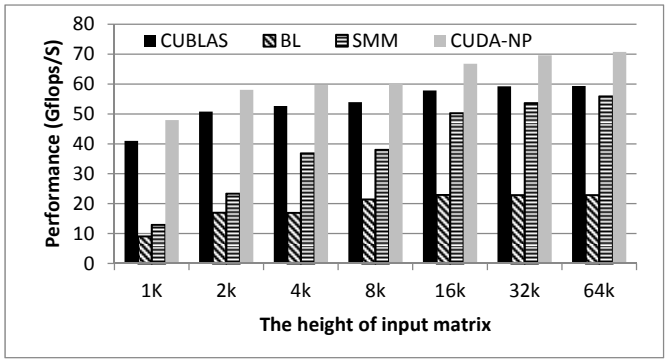

Figure 14. MV results on GTX 680 for matrices with variable heights and a constant width (2k).

We compare our result of MV to CUBLAS and SMM in Figure 14. The shared memory configuration is set to 48KB per SMX. We fix the width of input matrix to 2k and vary the height from 1k to 64k. The height determines the overall number of threads for the baseline. From Figure 14, we can see that our solution always outperforms both SMM and CUBLAS.

As discussed in Section 3.3, for a local memory array, we may replace it with an array in global memory/shared memory or partition it so that the small partitions can be allocated as registers. Among all benchmarks, we are able to apply such optimizations to LE and LIB. We show the performance results when using global memory, shared memory or register file to replace the local memory arrays for these two benchmarks in Figure 15. As show in the figure, using global memory does not help the performance, as the global memory is off-chip memory and the local memory can be cached through L1 cache. The performance of using shared memory depends on the usage of shared memory. Since the size of the local memory array of LE is about twice of the size of local memory array of LIB, we can see heavy shared memory usage for LE actually hurts the performance while we can observe the performance gains for LIB. Since the register file is much larger than shared memory, it works the best for both benchmarks.

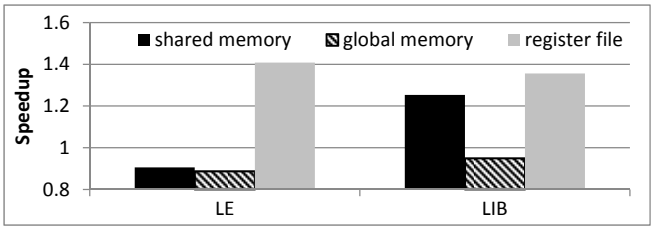

Figure 15. Comparing different ways to replace a local memory array.

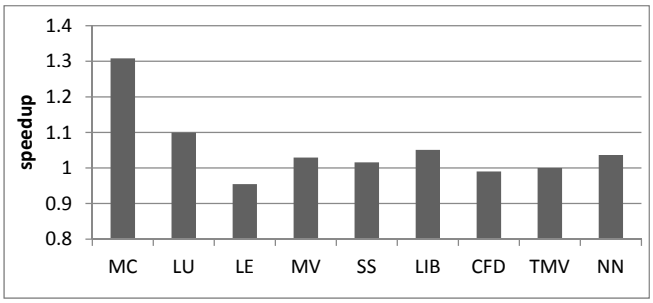

Figure 16. Speedup of using the shfl instruction over using shared memory for reduction or scan operations when applying intra-warp NP. The baseline is the best performing inter-warp NP version.

The benefit of using the __shfl instruction is shown in Figure 16. In the figure, we use the best inter-warp NP version as the baseline to be normalized to. Then we show the speedup of the version of using __shfl instruction for reduction or scan operations over the version of using shared memory. From this figure, we can see that the __shfl instruction is very useful for MC and LU. These two benchmarks have intensive shared memory usage, and using shared memory for reduction or scan operations can make it even more intensive. For other benchmarks, the impact of using __shfl instruction is minor as the scan or reduction primitive only takes a small amount of execution time. We also see some small slowdowns, and consider the reason may be due to memory behaviour changes.

6. Related Work And Discussion

Many compiler frameworks [3][21][23][38][41] have been developed to utilize high-throughput GPGPU architecture. For example, OpenMPC [21] is proposed to transfer legacy OpenMP programs to GPGPU programs, and [41] translates the naïve GPU programs into optimized ones. Most of these works focuses on optimizing global memory accesses and enabling massive thread-level parallelism. To the best of our knowledge, nested parallelism has been overlooked and our solution is the first compiler approach to exploit it.

It is shown in [10] that using dynamic parallelism can improve the performance of the divide-and-conquer algorithms as the child kernels can be launched by the GPU instead of the CPU. However, they also observed slowdowns due to dynamic parallelism for regular applications such as K-mean.

Dynamic parallelism is more suitable for massive nested loops instead of benchmarks listed in the paper. Furthermore, dynamic parallelism may require additional development effort due to the communication between a child kernel and a parent kernel. For example, shared memory is used in the nested loops for benchmarks, MC, LU, MV, SS, and BK. In order to utilize dynamic parallelism, developers have to write the code to copy the data from shared memory to global memory so that the child kernel can access the data, and then copy the data from global memory back to shared memory. Such expensive memory copy introduces both performance overhead and code In comparison, our CUDA-NP solution eliminates such redundant communications. For the benchmarks, NN, LE, LIB, and CFD, we implemented dynamic parallelism versions, which show 28.92, 7.61, 13.45, 125.67 and 52.29 times slower than their original versions, respectively, due to the communication cost and the dynamic-parallelism-enabled kernel overhead. Using NN as an example, which has a parallel loop with a large loop count, if we choose to let each thread of parent kernel launch a child kernel to perform the parallel loop, the dynamic parallelism version is about 28.92 times slower, compared to the version without dynamic parallelism. Then, we manually optimize it by only using the first thread in a TB to start a child kernel to reduce the number of kernel launches. This version is still 3.25 times slower. Overall, for benchmarks used in this paper, dynamic parallelism cannot help the performance for benchmarks even with manual optimizations, as the available NP is too limited to offset the high overhead of dynamic parallelism.

Some recent works focus on identifying the best performing version in a large search space, either using analytical models [14] or auto-tuning [28]. Since CUDA-NP only

generates a small number of versions, the optimal version can be found by testing these versions exhaustively. In other words, a simple auto-tuning can be used to find the optimal configuration. Furthermore, our experiments also reveal some key factors to find the optimal version for CUDA-NP. First, memory coalescing and intra-warp divergence can be used to determine the priority between intra-warp NP and inter-warp NP. Second, using 3 or 7 slave threads achieves close-to-optimal performance for all benchmarks in our study.

While many architectural improvements [24][28][35] and application optimizations [12][14][16][18] [29][31][32] [33][34][39][40][42][43][44] have been proposed for GPGPU programs, only few observe the potential of nested TLP in GPU programs. In [35] a hardware approach is proposed to create threads dynamically in the runtime to reduce the overhead of complex control flow, similar to Nvidia dynamic parallelism. In [14], a warp is used handle one job instead of one thread so as to reduce control divergence. However, in this paper, we show that inter-warp NP, i.e., distributing one master thread's workload into different warps, may be suitable for many benchmarks. Furthermore, compared to [14], our approach only requires an application developer to write pragmas instead of developing new programs. In addition, our compiler can also handle the reduction and scan variables and leverage the __shfl instructions.

Nested parallelism is specified as an option in the OpenMP and is supported in some implementations such as [22]. Compared to previous OpenMP studies [1][5][8][9] [10][11][13][36][37] on nested parallelism for CPUs, our solution is the first language extension to support nested parallelism for GPGPU platforms, and we show different ways to handle the scan and reduction operations, the inter-warp and intra-warp NP schemes, as well as careful resource managements on GPGPUs.

7. Conclusion

In this paper, we propose a novel compiler solution to leverage nested parallelism for GPGPU application. We observe that many benchmarks have relatively light nested parallelism, i.e., parallel loops with small loop counts, which cannot be effectively exploited using the recently introduced dynamic parallelism. Therefore, we propose to partition the code sections into sequential sections and parallel sections, and then enable different numbers of threads for sequential sections and parallel sections. In order to simplify application development, we implement our approach as a compiler framework to support directive-based nested parallelism. In our proposed CUDA-NP compiler framework, an application developer only needs to add OpenMP-like pragmas for parallel loops, and our compiler will generate the optimized code. Our compiler can handle the reduction and scan variables, chooses either the intra-warp NP or inter-warp NP approach to distribute the parallel loop iterations, and automatically handle different types of live variables across code sections. Our performance results show significant performance gains and demonstrate the effectiveness of our proposed solution.

Acknowledgments

We thank the anonymous reviewers for their insightful comments to improve our paper. This work is supported by an NSF grant CCF-1216569 and a NSF CAREER award CCF-0968667. We also want to thank the ARC Cluster [16] for providing Nvidia K20c GPUs.

References

[1] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. In TPDS, 2009.

[2] A. Bakhoda, G. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. Analyzing CUDA workloads using a detailed GPU simulator. In ISPASS April 2009.

[3] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. A Compiler Framework for Optimization of Affine Loop Nests for GPGPUs. In ICS, 2008.

[4] M. Boyer, D. Tarjan, S. T. Acton, and K. Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. In IPDPS 2009.

[5] HM. Bücker, A.Rasch, and A. Wolf. A class of OpenMP applications involving nested parallelism. In Proceedings of the 2004 ACM symposium on Applied computing, 2004.

[6] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In IISWC 2009.

[7] S. Collange, D. Defour, and Y. Zhang. Dynamic detection of uniform and affine vectors in GPGPU computations. In ICPP, 2009.

[8] L. Dagum, and R. Menon. OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, 1998.

[9] VV. Dimakopoulos, EH. Panagiotis, and GC. Philos.A microbenchmark study of OpenMP overheads under nested parallelism. In OpenMP in a New Era of Parallelism, 2008.

[10] J. DiMarco, and M. Taufer. Performance impact of dynamic parallelism on different clustering algorithms. In SPIE Defense, Security, and Sensing. International Society for Optics and Photonics, 2013.

[11] A. Duran, M. Gonzàlez, and J. Corbalán.Automatic thread distribution for nested parallelism in OpenMP. In ICS, 2005.

[12] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In Proc. Supercomputing, 2008.

[13] PE. Hadjidoukas, and VV. Dimakopoulos . Nested parallelism in the OMPI OpenMP/C compiler. In Euro-Par Parallel Processing, 2007.

[14] S. Hong and H. Kim. An analytical model for GPU architecture with memory-level and thread-level parallelism awareness. In Proc. International Symposium on Computer Architecture, 2009.

[15] S. Hong, S.K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In PPoPP 2011.

[16] http://moss.csc.ncsu.edu/~mueller/cluster/arc/

[17] B. Jang, D. Schaa, P. Mistry and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. In IEEE TPDS, 2010.

[18] O. Kayiran, A. Jog, M. T. Kandemir, C. R. Das. Neither More Nor Less: Optimizing Thread-level Parallelism for GPGPUs. In PACT, 2013.

[19] J. Kim, H. Kim, J. Lee, and J. Lee. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. In PPoPP, 2011.

[20] S. I. Lee, T. Johnson, and R. Eigenmann. Cetus – an extensible compiler infrastructure for source-to-source transformation. In LCPC, 2003

[21] S. Lee, S.-J. Min, and R. Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. In Proc. In PPoPP, 2009

[22] C. Liao, O. Hernandez, B. Chapman, W. Chen and W. Zheng. OpenUH: An Optimizing, Portable OpenMP Compiler. In the 12th Workshop on Compilers for Parallel Computers, Spain, 2006.

[23] Y. Liu, E. Z. Zhang, amd X. Shen. A Cross-Input Adaptive Frame-work for GPU Programs Optimization. In IPDPS, 2009.

[24] V. Narasiman, C. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In MICRO, 2011.

[25] Nvidia CUDA Toolkit 5.0 CUBLAS Library, 2013

[26] Nvidia GPU Computing SDK 5.0, 2013.

[27] Nvidia Programming Guide, CUDA Toolkit V5.5, 2013.

[28] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable performance on heterogeneous architectures. In ASPLOS, 2013.

[29] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen and W. Schulte. SIMD Parallelization of Applications that Traverse Irregular Data Structures. In CGO, 2013.

[30] T. G. Rogers, M.Connor, T. Aamodt, Cache-Conscious Wavefront Scheduling. In MICRO , 2012.

[31] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S. Ueng, J. A. Stratton, and W. W. Hwu. Optimization space pruning for a multi-threaded GPU. In CGO, 2008.

[32] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In PPoPP, 2008.

[33] G. Ruetsch and P. Micikevicius, Optimize matrix transpose in CUDA. Nvidia, 2009.

[34] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, A Performance Analysis Framework for Identifying Performance Benefits in GPGPU Applications. In PPoPP, 2012.

[35] M. Steffen and J. Zambreno. Dynamic Thread Creation for Improving Processor Utilization on SIMT Streaming Processor Architectures. In MICRO, 2010.

[36] Y. Tanaka, K. Taura, M. Sato, and A. Yonezawa. Performance evaluation of OpenMP applications with nested parallelism. In Languages, Compilers, and Run-Time Systems for Scalable Computers, 2000.

[37] X. Tian, JP. Hoeflinger, G. Haab, Y.K. Chen, M. Girkar, and S. Shah. A compiler for exploiting nested parallelism in OpenMP programs. Parallel Computing, 2005.

[38] S. Ueng, M. Lathara, S. S. Baghsorkhi, and W. W. Hwu. CUDA-lite: Reducing GPU programming Complexity, In LCPC, 2008

[39] V. Volkov and J. W. Benchmarking GPUs to tune dense linear algebra. In Proc. Supercomputing, 2008.

[40] B. Wu, Z. Zhao, E. Zhang, Y. Jiang, and X. Shen. Complexity Analysis and Algorithm Design for Reorganizing Data to Minimize Non-Coalesced GPU Memory Accesses. In PPoPP, 2013.

[41] Y. Yang, P. Xiang, J. Kong and H. Zhou. A GPGPU Compiler for Memory Optimization and Parallelism Management. In PLDI, 2010.

[42] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou. Shared Memory Multiplexing: A Novel Way to Improve GPGPU Throughput. In PACT, 2012.

[43] Y. Zhang, J. Cohen, and J. D. Owens. Fast Tridiagonal Solvers on the GPU. In PPoPP, 2010.

[44] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly elimination of dynamic irregularities for GPU computing. In ASPLOS, 2011.