Runtime Dependence Computation and Execution of Loops on Heterogeneous Systems

Jayvant Anantpur R. Govindarajan

Supercomputer Education and Research Centre
Indian Institute of Science
jayvant@hpc.serc.iisc.ernet.in govind@serc.iisc.ernet.in

Abstract

GPUs have been used for parallel execution of DOALL loops. However, loops with indirect array references can potentially cause cross iteration dependences which are hard to detect using existing compilation techniques. Applications with such loops cannot easily use the GPU and hence do not benefit from the tremendous compute capabilities of GPUs.

In this paper, we present an algorithm to compute at runtime the cross iteration dependences in such loops. The algorithm uses both the CPU and the GPU to compute the dependences. Specifically, it effectively uses the compute capabilities of the GPU to quickly collect the memory accesses performed by the iterations by executing the slice functions generated for the indirect array accesses. Using the dependence information, the loop iterations are levelized such that each level contains independent iterations which can be executed in parallel. Another interesting aspect of the proposed solution is that it pipelines the dependence computation of the future level with the actual computation of the current level to effectively utilize the resources available in the GPU. We use NVIDIA Tesla C2070 to evaluate our implementation using benchmarks from Polybench suite and some synthetic benchmarks. Our experiments show that the proposed technique can achieve an average speedup of 6.4x on loops with a reasonable number of cross iteration dependences.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Processors-Code Generation—Compilers

General Terms Algorithms, Languages, Performance

CGO'13 23-27 February 2013, Shenzhen China.
978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

1. Introduction

The compute capabilities of Graphics Processing Units (GPUs) [10][14] have significantly increased with each newer version. GPUs have been effectively used by programmers to speedup data parallel computations either by manually identifying loops to parallelize or using compilation techniques to identify and parallelize loops. There has been extensive research in the field of parallelizing techniques for loops [3][4][5][15][17][20]. Also optimization of loops for GPGPUs has been explored in [1][2][9]. But these techniques can not be used to parallelize loops with indirect array accesses for execution on GPUs.

Indirect array accesses are very effective in reducing memory consumption in sparse matrix applications. The inability of existing compilers to parallelize loops with indirect memory accesses, prevents the use of GPUs for accelerating such loops. The most common use of indirect array access is *A[B[i]]* where *i* is the loop index. Since the contents of array *B* are not known at compile time, the compiler can not make any assumptions about the accesses to array *A* and hence conservatively computes the data dependences on array *A*. This causes extra memory dependences to be added among instructions. For the example shown below, the automatic parallelizing compilers will assume cross iteration dependences in the loop and hence will not parallelize it.

```
for (i = 0; i < N; i++) {
    Arr1[ind1[i]] = Expr1;   //S1
    Arr2[i] = Arr1[ind2[i]]; //S2
}
```

In this example, static analysis will assume that there is a Read After Write dependence between S1 and S2, Write After Write dependence between two instances of S1 and Write After Read between S2 and S1.

In this paper we propose an algorithm to compute the dependences using both the CPU and the GPU and schedule at runtime the iterations of the loop in parallel honoring the dependences. The central idea is to levelize the iterations according to the memory dependences. Levelization basically generates a topological sort of the iterations using memory dependences and iteration indices to decide the execution or-

der. We effectively use the compute capabilities of the GPU to gather information of indirect array accesses. Using slice functions for the indices of conflicting array accesses, writes to arrays with conflicting accesses are logged, e.g., the value of *ind1[i]* and the iteration index *i*. When another conflicting access to the same location by another iteration is detected, our algorithm moves the iteration with higher iteration index to the next level. This way all the iterations of the loop are assigned levels. We refer to this process of assigning levels as levelization and this phase of our algorithm as Dependence Computation phase. Iterations at level *k+1* can be executed only after iterations at level *k* are executed. Further an iteration at level *k* can be executed independent of the others at the same level. We run the loop on the GPU by making one kernel call per level such that each kernel invocation executes only the iterations at that level. This way we can run multiple iterations of the loop in parallel without violating any cross iteration dependences.

Our algorithm can be software pipelined to compute the dependences at level *k+1* while the execution of level *k* is performed on the GPU. The algorithm has been designed to reduce the space overheads of dependence computation.

The main contributions of the paper are:

- Use of the compute capabilities of GPUs to identify the independent iterations in a loop and levelize them based on the dependences, for parallel execution.
- Execution of the levelized iterations on a GPU, by running all the iterations at the same level in parallel.
- Pipelining the dependence computation and execution phases to effectively use the compute capabilities of GPUs.

To the best of our knowledge, our work is the first to use a GPU to levelize the iterations of a loop with cross iteration dependences due to indirect array accesses and execute them in parallel on the GPU honoring the dependences.

2. Background

2.1 NVIDIA GPUs

In this section we describe the architecture of NVIDIA GPUs in general. They consist of a set of streaming multiprocessors (SMs) and each of these SMs has several scalar units. The Tesla C2070 has 14 SMs and each SM has 32 scalar cores making the total number of CUDA cores 448. Each SM has 64 KB of configurable shared memory and L1 cache. It can be configured as 48 KB of shared memory and 16 KB of L1 cache or as 16 KB of shared memory and 48 KB of L1 cache. It also has a 768 KB unified L2 cache. The L2 cache services all load, store and texture requests. In addition, C2070 supports 6 GB of global memory without ECC and 5.25 GB memory with ECC. The global memory can be accessed by all the CUDA cores and it provides a very high bandwidth, if the accesses are coalesced. However latency of accessing the global memory is high and in the order of a few hundreds of cycles.

2.2 CUDA Execution Model

Compute Unified Device Architecture (CUDA) [13] provides extensions to C and C++ languages, which can be used to define functions called *Kernels*. A kernel is executed in parallel using a set of parallel threads which are organized in thread blocks and grids of thread blocks. Grids and thread blocks are arranged as three dimensional structures. Each thread executes an instance of the kernel and has a unique thread ID. All the threads in a thread block can synchronize using *__syncthreads* intrinsic function, which acts as a barrier which can be crossed only after all the threads in the block have reached the barrier. Threads within a thread block can share data using the shared memory provided on the SMs. Kernel calls execute asynchronously in the sense that control is returned to the calling CPU thread before the kernel execution on the GPU completes.

Modifications to the global memory done by one thread are not immediately visible to other threads. CUDA provides various memory fence functions which can be used to block a thread till its global and shared memory updates are visible to other threads in its block and all the threads in the device. Another way to achieve the necessary synchronization and memory consistency to satisfy Read After Write dependences is to execute the writes and reads in two different kernels. CUDA also provides functions to perform read-modify-write operations atomically.

3. Dependence Computation and Execution

In this section we describe our algorithm to levelize iterations of a loop with cross iteration dependences due to indirect array accesses and to execute the iterations in parallel on a GPU.

3.1 Overview

We use the following code as the running example.

```
for (i = 0; i < N; i++) {
    int wIdx = ind1[i];
    Arr1[wIdx] = Expr1;   //S1
    int rIdx = ind2[i];
    Arr2[i] = Arr1[rIdx]; //S2
}
```

In this example there are indirect accesses to array *Arr1*. Since these accesses can be potentially conflicting, i.e., two iterations may access the same array element and at least one of them is a write access, we consider such arrays as arrays with conflicting accesses. We are interested in monitoring accesses only to such arrays. Accesses to array *Arr2* are data parallel i.e., no two iterations will access the same element of array *Arr2*. We assume that the arrays *ind1* and *ind2* are not modified in the loop and arrays *Arr1* and *Arr2* are non overlapping. Our approach uses the proposed runtime technique only in places where compiler analysis cannot resolve the dependences.

If for two different values of *i*, say *i1* and *i2*, where *i1* < *i2*, *wIdx* has the same value, then there is a Write After Write (WAW) dependence between the corresponding instances of statement S1. If the value of *wIdx* in iteration *i1* is the same as the value of *rIdx* in iteration *i2*, then there is a Read After Write (RAW) dependence (*S2* depends on *S1*), whereas, if the value of *rIdx* in iteration *i1* is the same as the value of *wIdx* in iteration *i2*, then there is a Write After Read (WAR) dependence (*S1* depends on *S2*). In all these three cases, to match the sequential execution semantics, iteration *i1* should be executed before iteration *i2*. So we can assign levels to the iterations such that iterations at level *k* are not dependent on each other and are dependent directly or indirectly on iterations at level *k-1, k-2, ... , 1*, which means iterations at level *k* can not be executed till all the iterations at the earlier levels have been executed. Thus if iteration *i1* is at level *k*, then *i2* should be at a level greater than *k*. Moreover, since all the iterations at the same level are independent of each other, they can be executed in any order.

To compute dependences, we are interested only in the values of *wIdx* and *rIdx*. We neither need to evaluate *Expr1* nor do we need to write to array *Arr1* or read from *Arr1*. To detect conflicting accesses we need to compare the values of the index variables *wIdx* and *rIdx* in each iteration with their values in every other iteration. Since the detection is performed at runtime, both the runtime and memory overheads of detecting conflicts should be as small as possible.

Algorithm 1 Levelization

1: **procedure** Levelize()
2: $S = set\_of\_all\_iterations$
3: $level \leftarrow 1$
4: **while** $NonEmpty(S)$ **do**
5: $\{S\_c, S\_f\} = Partition(S)$
6: $AssignLevel(S\_c, level)$
7: $level \leftarrow level + 1$
8: $S \leftarrow S\_f$
9: **end while**
10: **end procedure**

Given a loop with potentially loop-carried dependences, our solution can be used to parallelize it. The phases involved are as follows:

- generate the data parallel CUDA version of the loop ignoring dependences.
- levelize the iterations by computing dependences.
- execute the iterations level by level, parallelizing all the iterations at each level.

The levelization algorithm partitions the set of iterations S into sets of iterations $S_1, S_2, ..., S_n$ such that all the S_i's are disjoint and their union is S. Further each set S_i corresponds to the iterations at level i which are independent and can be executed in parallel. At each step the algorithm divides the set of remaining iterations into two disjoint sets viz., S_c and S_f such that each iteration is either in S_c or S_f. All the iterations in S_c can be executed at the current level *i* and are independent of each other; all the iterations in S_f are at future levels i.e., level *i+1* or above. The partioning step is then carried out on the set S_f as described in Algorithm 1. Thus our algorithm partitions the input set of iterations one level at a time to reduce memory overheads. After the levelization algorithm assigns levels to iterations, the iterations can be executed level by level and all the iterations at each level in parallel.

The method for generating the data parallel code is described in Section 3.2.

Our algorithm currently targets innermost loops but can also be used to parallelize outer loops where inner loops are run sequentially.

3.2 Generation of Data Parallel Code

The data parallel CUDA version of the loop is generated ignoring the dependences due to indirect memory accesses. This makes the generated kernel unsafe to run on the GPU. The data parallel code can be generated by converting the iteration index to tIdx and making the computation a single dimension grid, and choosing the execution configuration appropriately. In our work we assume that the data parallel code is also given by the user. For our running example, the generated CUDA code is as shown in kernel function *loopKernel*. The kernel is invoked with as many threads as the number of iterations.

```
#define KERNEL_PARAMS \
int* Arr1, int* Arr2, int* ind1, int* ind2, int N

__global__ void loopKernel(KERNEL_PARAMS) {
  int tIdx = blockIdx.x * blockDim.x + threadIdx.x;
  if (tIdx >= N) return;
  int wIdx = ind1[tIdx];
  Arr1[wIdx] = Expr1;
  int rIdx = ind2[tIdx];
  Arr2[i] = Arr1[rIdx];
}
```

3.3 Levelization

Now we will describe the levelization process in detail. It consists of a Dependence Computation Phase which is further divided into a Writer Phase and a Reader Phase. Algorithms 2, 3 and 4, describe these phases in detail.

3.3.1 Dependence Computation Phase

From the kernel definition, we can identify arrays with potentially conflicting accesses. Using the accesses to such arrays as the starting point we can extract the slice functions for the indices of their accesses. In our example, the array with conflicting accesses is *Arr1* and the indices are *wIdx* and *rIdx*. The Dependence Computation Phase needs to log the iteration numbers and indices of *Arr1* that are written to. For this purpose an auxiliary array of size less than or equal to the size of the original array is used. In this example, we will assume that the auxiliary array is of the same size as the

original array. From the original kernel two kernels are generated, one to log into the auxiliary array the indices that are written to and another to check the indices that are read. Both these kernels use the slice functions for the indices that are accessed. In addition to the auxiliary array, we use an array, *Levels*, to store the level of each iteration. All the iterations are initially assumed to be at level *1*.

Algorithm 2 DepCompWriter

```
1: procedure checkWr (iter, arrIdx, curLvl)
2: flag ← false
3: curSmallestWr ← getCurSmallestWr(arrIdx)
4: if iter < curSmallestWr then
5:    incrLevel(curSmallestWr) {/*WAW*/}
6:    setCurSmallestWr(arrIdx, iter)
7:    flag ← true
8: else if iter > curSmallestWr then
9:    iterLvl ← getIterLevel(curSmallestWr)
10:   if iterLvl < curLvl then
11:      setCurSmallestWr(arrIdx, iter)
12:   else
13:      incrLevel(iter) {/*WAW*/}
14:      flag ← true
15:   end if
16: end if
17: return flag
18: end procedure
19:
20: procedure DepCompWriter(curLvl, Arr)
21: iter ← tIdx
22: flag ← false
23: iterLvl ← getIterLevel(iter)
24: if iterLvl = curLvl then
25:    writtenIndices ← getWrittenIndices(Arr, iter)
26:    for all arrIdx ∈ writtenIndices do
27:       flag ← flag ∨ checkWr(iter, arrIdx, curLvl)
28:    end for
29: end if
30: return flag
31: end procedure
```

3.3.2 Writer Phase

Algorithm 2 iterates over all the locations of the array with conflicting accesses, that are written to and checks for WAW dependences using the procedure *checkWr*, which stores the smallest iteration index that writes to each location. We use CUDA's *atomicMin* to retain the smallest iteration index. Function *setCurSmallestWr* stores in the auxiliary array the smallest iteration index that writes to the input location *arrIdx*, whereas the function *getCurSmallestWr* returns that index. When two different iterations write to the same location, then there is a WAW dependence and so the iteration with the larger index is pushed to the next level if the iteration with the smaller index is not at a lower level. This way the two conflicting iterations will not execute at the same level. The value returned by the procedure *DepCompWriter* indicates whether any iteration got pushed to the next level. Function *incrLevel* increments the level of the input iteration index by 1 and function *getIterLevel* returns the level of the input iteration index. Function *getWrittenIndices(Arr, iter)* returns indices of array *Arr*, that are written to by the iteration *iter*.

Algorithm 3 DepCompReader

```
1: procedure checkRd (iter, arrIdx, curLvl)
2: flag ← false
3: curSmallestWr ← getCurSmallestWr(arrIdx)
4: if iter < curSmallestWr then
5:    incrLevel(curSmallestWr) {/*WAR*/}
6:    flag ← true
7: else if iter > curSmallestWr then
8:    iterLvl ← getIterLevel(curSmallestWr)
9:    if iterLvl < curLvl then
10:      setCurSmallestWr(arrIdx, UNINT)
11:   else
12:      incrLevel(iter) {/*RAW*/}
13:      flag ← true
14:   end if
15: end if
16: return flag
17: end procedure
18:
19: procedure DepCompReader(curLvl, Arr)
20: iter ← tIdx
21: flag ← false
22: iterLvl ← getIterLevel(iter)
23: if (iterLvl = curLvl) ∨ (iterLvl = curLvl + 1) then
24:    readIndices ← getReadIndices(Arr, iter)
25:    for all arrIdx ∈ readIndices do
26:       flag ← flag ∨ checkRd(iter, arrIdx, curLvl)
27:    end for
28: end if
29: return flag
30: end procedure
```

3.3.3 Reader Phase

Algorithm 3 detects WAR and RAW dependences. It iterates over all the locations of the array with conflicting accesses that are read and checks for WAR and RAW conflicts using the procedure *checkRd*. If the reader iteration index is smaller than the writer iteration index, then there is a WAR dependence and hence the writer iteration is pushed to the next level, whereas, if the reader iteration index is greater than the writer iteration index and the writer iteration is not at a lower level, then there is a RAW dependence and so the reader iteration is pushed to the next level. The return value indicates whether any iteration got pushed to the next level. Function *getReadIndices(Arr, iter)* returns indices of array *Arr*, that are read by the iteration *iter*.

The top level algorithm to compute dependences is shown in Algorithm 4. It finds iterations at each level, one level at a time. The while loop will run as many times as there are levels in the iterations of the input loop. Each iteration of the

Algorithm 4 Dependence Computation

```
1: procedure DepComp()
2: moreLvls ← true
3: curLvl ← 1
4: while moreLvls ≠ false do
5:     moreLvls ← DepCompWriter(...)
6:     moreLvls ← moreLvls ∨ DepCompReader(...)
7:     curLvl ← curLvl + 1
8: end while
9: end procedure
```

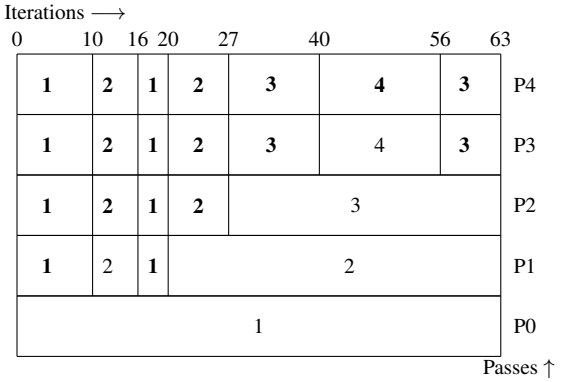

Figure 1: Levelization Steps. The numbers on the top show the iteration numbers, P0 shows the initial level values, P1 to P4 show the level values at the end of each pass of Dependence Computation Phases, Bold numbers indicate levels have been identified.

while loop is referred to as a pass. At the end of the while loop, each iteration will have been assigned a level value based on its accesses to the arrays with conflicting accesses. Calls to procedures *DepCompWriter* and *DepCompReader* are executed on the GPU with as many threads as the number of iterations. The procedure *DepComp* executes on the CPU.

Figure 1 shows an example of how iterations get levelized. The numbers inside the boxes show the level numbers. Pass P0 shows the levels at the beginning of the Dependence Computation Phase. P1 to P4 show the level values after the end of passes 1 to 4 of the Dependence Computation Phase. As shown in the figure, at the end of pass P1, iterations 0-10 and 16-20 are at level 1, iterations 11-15 and iterations 21-63 are at level 2. Pass P1 computes memory access dependences for all the 64 iterations. Pass P2 computes dependences for iterations 11-15 and 21-63. At the end of pass P2, iterations 27-63 are at level 3. Pass P3 finds that out of 27-63, iterations 40-56 are at level 4 and they remain at level 4 at the end of pass P4. Since there is no change in the levels at the end of pass P4, the algorithm stops.

3.4 Execution Phase

Once the Dependence Computation Phase completes, the iterations are then executed level by level. We call this phase the Execution Phase. In this phase a modified version of the original kernel is executed as many times as the number of levels found by the Dependence Execution Phase. The algorithm is shown in Algorithm 5

Algorithm 5 ExecPhase

```
1: procedure ExecPhase()
2: level ← 1
3: while level ≤ totalNumLevels do
4:     loopKernelExec(...)
5:     level ← level + 1
6: end while
7: end procedure
```

The kernel *loopKernelExec* is the modified kernel generated from the original kernel *loopKernel*. The only difference between them is the level comparison check. This check ensures that the iterations are executed in a levelized manner which in turn ensures that any dependence due to indirect accesses is always satisfied. Since an iteration at level *k+1* has a memory dependence on some iteration at level *k*, all the iterations at level *k* should finish before any iteration at level *k+1* can start executing. One way of ensuring this is to make a separate kernel call for each level. We used this mechanism because of its simplicity. Other mechanisms to achieve this synchronization involve more changes to the code and grid structure.

```
__global__ void loopKernelExec(KERNEL_PARAMS,
                      int* Levels, int level) {
  int tIdx = blockIdx.x * blockDim.x + threadIdx.x;
  if (tIdx >= N) return;
  if (Levels[tIdx] != level) return; //level check
  int wIdx = ind1[tIdx];
  Arr1[wIdx] = Expr1;
  int rIdx = ind2[tIdx];
  Arr2[i] = Arr1[rIdx];
}
```

3.5 Optimizations

In this section we will discuss various optimizations to reduce memory and runtime overheads of the algorithm discussed in the previous sections.

3.5.1 Static Chunking

In the previous discussion we had assumed that the size of the auxiliary array is the same as the size of the array with conflicting accesses. For benchmarks with large arrays, this may increase the memory requirements beyond the size of the global memory available on the GPU. To reduce the memory overheads we modified the algorithm to work with smaller auxiliary arrays. The algorithm described in the previous section, indexed the auxiliary array with the index of the array with conflicting accesses. For smaller sizes of auxiliary arrays, we changed the indexing mechanism to use a simple hash function. In our implementation we used the modulo function as the hash function.

Using the hash function may cause two or more indices to map to the same location in the auxiliary array and hence can cause false conflicts among the array accesses. To reduce the effect of such false conflicts, the iteration space is divided into chunks, such that the iterations in chunk *N* are initialized

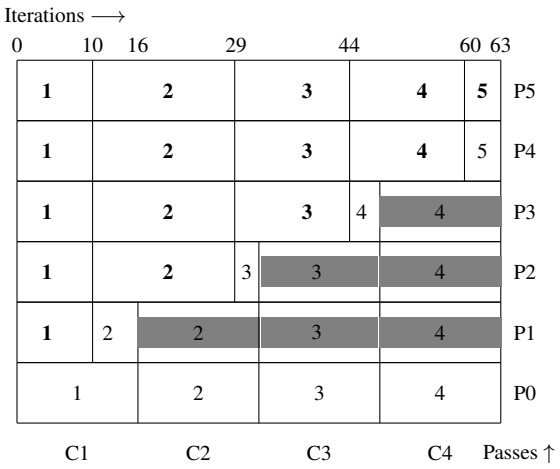

Figure 2: Static Chunking. The numbers on the top show the iteration numbers, P0 shows the initial level values, P1–P5 show the level values at the end of each pass of Dependence Computation Phase, C1–C4 are the chunk numbers, Bold numbers indicate levels have been identified.

to level *N*. The number of chunks is decided by the ratio of the sizes of the array with conflicting access and the corresponding auxiliary array. The dependence computation itself would be performed in *N* or more number of passes. In other words, the *k*th chunk will be processed in pass *k*. The advantage of static chunking is that it limits the number of iterations processed in each pass. For example, in the first pass of dependence computation, only iterations from the first chunk are processed, in the second pass iterations from the first chunk which are at level 2 and iterations from the second chunk are processed and so on.

Figure 2 shows an example of static chunking. Initially the iterations are divided in 4 chunks, each of size 16 such that iterations in chunk 1 are at level 1, iterations in chunk 2 are at level 2 and so on. The first pass of Dependence Computation Phase i.e. pass P1 evaluates only iterations from chunk 1 and iterations 10-15 are pushed to level 2. So in pass P1 iterations from chunk 2, 3 and 4 are not processed. This is shown by gray color in the corresponding boxes. Pass P2 processes iterations 10-31. This example shows that each pass processes only a fraction of the iterations as compared to the approach shown in figure 1. This also helps reduce the number of memory accesses to auxiliary array, levels array etc. Thus static chunking can reduce the cost of each pass but can potentially increase the number of levels.

3.5.2 Dynamic Chunking

When there are lot of conflicting accesses among the iterations, large number of iterations get pushed to the next level. An iteration needs to be considered for Dependence Computation till its level does not increase. This means if an iteration gets pushed *k* times, it will be part of Dependence Computation Phase *k+1* times. In case of a loop with a lot of conflicting accesses, many of the iterations will be pushed to the next level. Static chunking helps in case of loops with few conflicting accesses which are far apart, but if the conflicting accesses are nearby, then we may see many levels in a chunk also. To reduce the number of conflicts, we dynamically create more levels. To be able to decide when to create more levels, we use a simple heuristic that counts the number of conflicts seen and the average distance between conflicting iterations. We use CUDA's *atomicAdd* primitive to compute these and to reduce the cost of these atomic operations, we sample some of the conflicts. When the number of conflicts is above a certain threshold, the dynamic chunking mechanism creates more levels in each chunk.

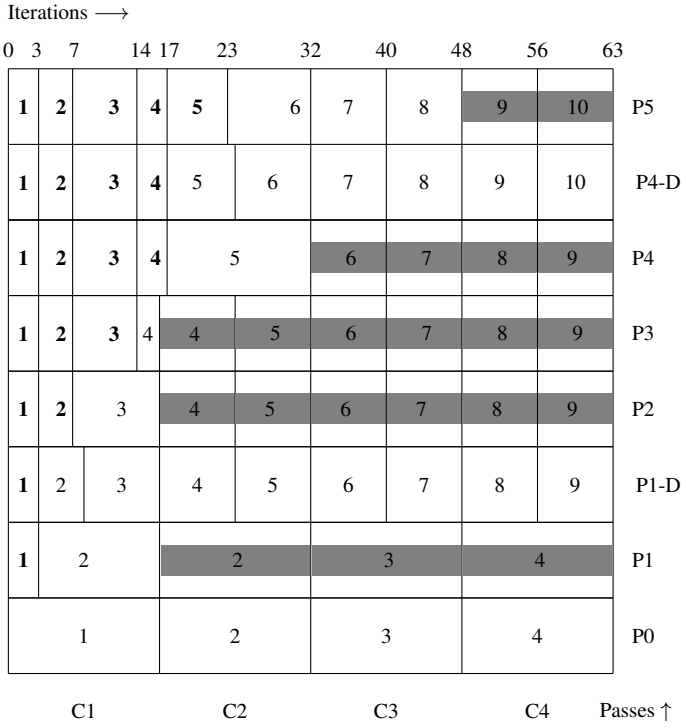

Figure 3: Dynamic Chunking. The numbers on the top show the iteration numbers, P0 shows the initial level values, P1–P5 show the level values at the end of each pass of Dependence Computation Phase, P1-D and P4-D show the dynamic chunking steps, Bold numbers indicate levels have been identified. Levelization only up to first 5 levels shown.

We explain the dynamic chunking mechanism with the help of an example. Assume we start with a static partitioning of iterations into 4 chunks, such that iterations in chunk *k* start with level *k*, and at the end of the first call to the Dependence Computation kernels, the number of conflicts is above the threshold, pushing a lot of iterations in the first chunk to level *2*, as shown in Figure 3. The boxes with gray color show the iterations that are not processed by the Dependence Computation kernels. The next call to the Dependence Computation kernels will have to consider all these iterations and also the iterations from the second chunk as they are also at level *2*. This may increase the number of conflicts seen in the second pass of Dependence Computation and hence a corresponding increase in the number of iterations going to level *3*. To avoid this, the algorithm creates two levels in each of the chunks, such that iterations in the second half of the chunk are at one level above the itera-

tions in the first half of the chunk. As shown in pass P1-D in Figure 3 the iterations in chunk *1* that are at level *1*, remain at that level, but the iterations that are at level *2* and are in the second half of the chunk are pushed to level *3*. Similarly, the second chunk will have iterations starting from levels *4* and *5*. This will roughly double the number of levels at the end of Dependence Computation Phase and hence roughly halve the number iterations considered in each pass. As the number of levels increases, the cost of kernel call invocations also increases and so the algorithm doubles the number of levels in each chunk only the first time the number of conflicts exceeds the threshold. In the subsequent passes, on encountering conflicts more than the threshold, the iterations in the later chunks are pushed by *1* to get a linear increase in the number of levels. For example, while computing dependences at level *4*, if the number of conflicts seen is above the threshold, then the levels of the iterations in chunk *3* and above are increased by one i.e. iterations in chunk *3* will go from levels *6* and *7* to *7* and *8* and so on as shown in pass P4-D. Also the levels of the iterations in the second half of chunk *2* are increased from *5* to *6*. This way the number of iterations considered for dependence computation in the next pass is reduced. Since this step of reassigning levels to iterations is completely data parallel, it is executed on the GPU.

If the average distance between conflicting iterations is very small and the number of conflicting iterations is very large, the number of levels will be large and so it may not make sense to run the loop on the GPU. The worst case is a sequential loop i.e. each iteration is dependent on its previous iteration. Our algorithm recognizes such cases early and suggests running the loop sequentially on the CPU.

3.5.3 Software Pipelining

In the original algorithm, the Execution Phase starts only after completion of the Dependence Computation Phase. Once the Dependence Computation Phase assigns a level to an iteration, that iteration does not take part in the Dependence Computation Phase again. We can make use of this property to pipeline the Execution Phase with the Dependence Computation Phase. In other words, when the iterations at level *k+1* are being considered for dependence computation, iterations at level *k* can be executed safely. Algorithm 6 shows the modified version of algorithm 2.

With the pipelining mechanism, by the time all the iterations are levelized, the iterations up to the second last level are executed. The iterations at the last level are then executed separately. If there are *n* levels, the total number of kernel calls in the original algorithm is *3n*. Out of these, *2n* calls are to the two dependence computation kernels *DepCompWriter* and *DepCompReader*, and *n* calls are to the Execution kernel. Software pipelining reduces the number of kernel calls by *n-1*, reducing the kernel call overheads.

Figure 4 shows an example of how pipelining of dependence computation and execution phases is achieved. In pass P2, when the dependence computation of level 2 iterations is being done, level 1 iterations are executed. Similarly, in pass P3, dependence computation of iterations at level 3 is overlapped with the execution of iterations in level 2.

Algorithm 6 DepCompWriterSWP

```
1: procedure DepCompWriterSWP(curLvl, Arr)
2: iter ← tIdx
3: flag ← false
4: iterLvl ← getIterLevel(iter)
5: if iterLvl = curLvl − 1 then
6:    loopKernelExec()
7: else if iterLvl = curLvl then
8:    writtenIndices ← getWrittenIndices(Arr)
9:    for all arrIdx ∈ writtenIndices do
10:      flag ← flag ∨ checkWr(iter, arrIdx, curLvl)
11:   end for
12: end if
13: return flag
14: end procedure
```

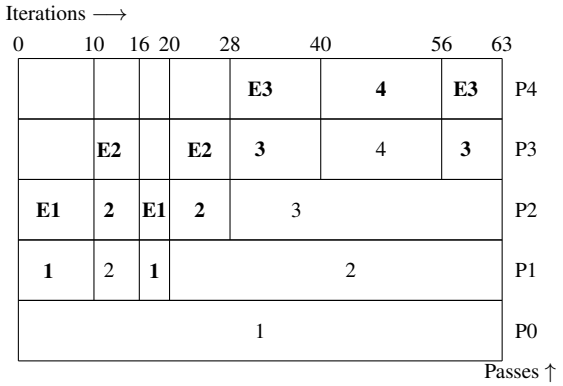

Figure 4: Software Pipelining of Dependence Computation and Execution Phases. The numbers on the top show the iteration numbers, P0 shows the initial level values, P1–P4 show the level values at the end of each pass of Dependence Computation Phase, Bold numbers indicate levels have been identified. E1–E3 indicate the execution phase.

One of the side effects of pipelining is the potential increase in memory and control divergences, as some threads in a warp may execute the dependence kernel code whereas others the execution kernel code. But for loops with nearby iterations at the same level, the effect is very small.

3.5.4 Dependence Computation Reuse

In cases where the loop with cross iteration dependences is an inner loop and the index arrays do not change from one iteration of the outer loop to other, as in the case of *fdtd-2d*, *Levels* array can be reused and the cost of Dependence Computation Phase can be avoided. Compile time analysis or user annotations can be used to identify which index arrays do not change. In our algorithm if reuse optimization is turned on, the *Levels* array is copied to the CPU at the end of the Dependence Computation Phase and from the second iteration of the outer loop, the Dependence Computation Phase is skipped and the *Levels* array is copied back from the CPU to the GPU. If the GPU has enough memory to store

Levels array then transferring it from the CPU to the GPU can be avoided increasing the benefit of the optimization.

4. Experimental Evaluation

We evaluate our algorithm by measuring the performance improvement over a sequential run on the CPU.

BM	Iters	Array Sizes	Kernels
randAcc	64M	64M	1
randAcc3	64M	64M	1
doAll	64M	64M	1
2dconv	64M	64M, 64M	1
fdtd-2d	64M	64.008M, 64.008M, 64M	3
gemm	1M	1M, 1M, 1M	1
mvt	4K	16M, 4K, 4K, 4K, 4K	2

Table 1: Description of benchmarks.

Table 1 shows the list of benchmarks and various parameters of the benchmarks. The column *Iters* shows the number of iterations in the loop to be parallelized. The column *Array Sizes* shows the sizes of the arrays which can potentially have conflicting accesses. *Kernels* shows the number of kernels (corresponding to loops to be parallelized) on which our algorithm was run. Benchmarks *2dconv*, *fdtd-2d*, *gemm* and *mvt* are from the Polybench [12] benchmark suite. Even though these benchmarks can be parallelized manually or using automatic compilation techniques, we assumed the arrays can have conflicting accesses. Hence their reads and writes are monitored. A similar approach was used by some of the recent work on speculative parallelization [8][16]. We ran our algorithm on these benchmarks to measure the performance with respect to sequential runs on the CPU, the benefits due to various optimizations, and overheads of dependence computation. The remaining three benchmarks viz., *randAcc*, *randAcc3* and *doAll* are synthetic benchmarks created using the benchmarks from the Polybench suite. *randAcc* and *randAcc3* have array access conflicts.

Benchmark *2dconv* has a kernel which computes *B[i][j]* using *A[i][j]* and its eight neighbors. In this benchmark we assume that there can be conflicting access to arrays *A* and *B* and so the writes to *B* and reads of *A* are monitored.

BM	Lvls	GPU Time			CPU Time	Speedup
		DC+Exec	Mem	Total		
randAcc	19	1.256	0.646	1.902	2.852	1.50
randAcc3	19	1.010	0.649	1.659	1.832	1.10
doAll	1	0.058	0.650	0.708	0.433	0.61
2dconv	1	0.094	0.467	0.561	0.479	0.85
fdtd-2d	1	1.869	0.397	2.266	3.916	1.73
gemm	1	0.135	0.075	0.210	7.910	37.66
mvt	1	0.048	0.090	0.138	0.195	1.41

Table 2: Speedup with auxiliary array size = array size, dynamic chunking on and pipelining on, DC = Dependence Computation, Mem = Memory transfer.

Benchmark *fdtd-2d* has 3 kernels. The first kernel reads from arrays *ey* and *hz* and writes to array *ey*. Our algorithm monitors the writes to *ey* and reads of both *ey* and *hz*. The second kernel reads from arrays *ex* and *hz* and writes to array *ex*. Hence the writes to *ex* and reads of both *ex* and *hz* are monitored. The third kernel reads from arrays *ex*, *ey* and *hz* and writes to array *hz*. The writes to *hz* and reads of all the 3 arrays are monitored. The three kernels are called in a loop. All the results except the one shown in Figure 4 are obtained with the number of iterations set to 10.

In benchmark *gemm*, writes to array *c* and reads from arrays *a* and *b* are monitored.

Benchmark *mvt* has two kernels and 5 arrays. Our algorithm monitors all the 5 arrays. The first kernel reads from arrays *a*, *y1* and *x1* and writes to array *x1*. The second kernel reads from arrays *a*, *y2* and *x2* and writes to array *x2*.

We used a system with an 8 core Intel Xeon processor running at 3.07GHz and Tesla C2070 GPU. We used nvcc version 4.0 for compiling both the CUDA and C code. For CUDA we used the default optimization level with L1 cache turned off and for compiling the C code we used -O3 optimization level. We ran each benchmark 5 times and took the average runtime. All the runtimes are in seconds. The execution time reported in the results section is for all the kernels in the benchmarks, but not the entire benchmark. However, it includes the memory allocation and transfer time.

For each benchmark, we need slice functions for the indirect array access indices. We hand generated the slice functions using the techniques described in [21]. We assumed that the slice functions do not have side effects. We also expect that the arrays with potentially conflicting accesses and their sizes are known at compile time. We assume the loop has already been translated to CUDA, ignoring the side effects of the indirect memory accesses and also an execution configuration for the kernel call has been provided. The input to our system is the names of the arrays with conflicting accesses, their sizes, the grid configuration, the slice functions, the original kernel and its formal and actual arguments. Our system generates CUDA code for the Dependence Computation and Execution kernels. It also generates a wrapper C function for calling the Dependence Computation and Execution kernels. The code for all the optimizations such as pipelining of the two phases, reuse of dependence computation and dynamic chunking is also generated. We replaced the original kernel call in the benchmarks by a call to the generated wrapper function.

Table 2 shows the results with the optimized mode of the algorithm. In this mode, the size of the auxiliary array is equal to the size of the array with conflicting accesses, dynamic chunking is turned on and pipelining of dependence computation and iteration execution is also on. We get speedups from 1.1x to 1.5x on the benchmarks with conflicting array accesses. As the table shows, the benchmarks can have as many as 19 levels. For the benchmarks with data parallel loops we see more speedup mainly because there are no conflicting accesses and hence the algorithm finds that all iterations can be run together. The benefits of our algorithm depend on the number of levels in the iterations and the size

of the slice function relative to the actual computation. For all the benchmarks we use, actual computation performed is comparable to the size of the slice function. This is one of the reasons for the moderate speedup achieved. Table 2 also shows the memory transfer time. In benchmarks *doAll* and *2dconv*, the memory transfer time is more than the sequential execution time of the loop on the CPU. In other benchmarks also, it is a considerable portion of the total GPU Time. This is another factor for the moderate speedup achieved. Table 3 shows the overhead of dependence computation as a fraction of the sequential runtime on the CPU.

BM	Lvls	GPU Time			CPU Time	DC/CPU
		DC	Exec+Mem	Total		
randAcc	19	0.963	1.024	1.987	2.852	0.34
randAcc3	19	0.793	0.950	1.743	1.832	0.43
doAll	1	0.031	0.677	0.708	0.433	0.07
2dconv	1	0.054	0.517	0.561	0.479	0.11
fdtd-2d	1	1.506	0.760	2.266	3.916	0.38
gemm	1	0.085	0.125	0.210	7.910	0.01
mvt	1	0.027	0.111	0.138	0.195	0.14

Table 3: Cost of Dependence Computation (DC) phase wrt CPU runtime. Pipelining is turned off to measure the time spent in Dependence Computation Phase and Execution Phase.

BM	Aux Arr	Lvls	CPU Time	GPU Time		Speedup	
				R-Off	R-On	R-Off	R-On
fdtd-2d	1	1	19.51	9.760	7.371	1.99	2.65
fdtd-2d	0.5	2	19.51	11.350	7.761	1.72	2.51
fdtd-2d	0.25	4	19.51	15.285	8.651	1.27	2.25
fdtd-2d	0.125	8	19.51	18.410	10.465	1.06	1.86

Table 4: Speedup with Reuse of Dependence Computation on (R-On) and off (R-Off), different auxiliary array sizes, dynamic chunking on, pipelining on. AuxArr = size of auxiliary array / size of user array rounded to the nearest power of 2.

Table 4 shows the improvement due to reuse of dependence computation. Benchmark fdtd-2d runs the kernels in a loop. The number of iterations is set to 50 for these measurements. When the reuse of dependence computation is turned on, the results of the Dependence Computation Phase are stored on the CPU at the end of the first execution of the loop and then copied back from the CPU to the GPU for the later executions of the loops. The speedup will be more if we store the Dependence Computation results on the GPU.

BM	Lvls	CPU Time	GPU Time		Speedup	
			P-Off	P-On	P-Off	P-On
randAcc	19	2.852	1.990	1.901	1.43	1.50
randAcc3	19	1.832	1.747	1.655	1.05	1.11

Table 5: Speedup with pipelining on (P-On) and off (P-Off), dynamic chunking on.

Table 5 shows the performance improvement with pipelining of Dependence Computation and Execution Phases. The gains are due to reduction in the number of kernel calls.

Table 6 shows the effect of reducing the auxiliary array sizes. As the size goes down from 1x of the conflicting array size to 1/8x, false conflicts increase with an increase in the number of levels and the dependence computation time. In our experiments we observed slowdowns of up to 2.25x due to reduction in the auxiliary array sizes.

BM	AuxArr	Lvls	Time		Speedup
			GPU	CPU	
randAcc	1	19	1.902	2.852	1.50
	0.5	24	2.094	2.852	1.36
	0.25	40	2.310	2.852	1.23
	0.125	64	2.577	2.852	1.11
randAcc3	1	19	1.659	1.832	1.10
	0.5	24	1.853	1.832	0.99
	0.25	40	2.047	1.832	0.89
	0.125	65	2.340	1.832	0.78
doAll	1	1	0.708	0.433	0.61
	0.5	2	0.718	0.433	0.60
	0.25	4	0.739	0.433	0.58
	0.125	8	0.770	0.433	0.56
2dconv	1	1	0.561	0.479	0.85
	0.5	2	0.562	0.479	0.85
	0.25	4	0.596	0.479	0.80
	0.125	8	0.612	0.479	0.78
fdtd-2d	1	1	2.266	3.916	1.73
	0.5	2	2.579	3.916	1.52
	0.25	4	3.182	3.916	1.23
	0.125	8	4.001	3.916	0.98
gemm	1	1	0.210	7.910	37.66
	0.5	2	0.212	7.910	37.31
	0.25	4	0.215	7.910	36.79
	0.125	8	0.218	7.910	36.28
mvt	1	1	0.138	0.195	1.41
	0.5	2	0.140	0.195	1.39
	0.25	4	0.202	0.195	0.96
	0.125	8	0.308	0.195	0.63

Table 6: Speedup with different auxiliary array sizes, dynamic chunking on and pipelining on. AuxArr = size of auxiliary array / size of user array rounded to the nearest power of 2.

5. Related Work

Recently there have been a few attempts to use GPUs to run DOACROSS loops. Paragon [16] identifies possibly-data parallel loops and runs them speculatively on the GPUs and also runs them sequentially on the CPU. In case of misspeculation, the data generated by the GPUs is ignored and the data generated by the CPU is used. Feng et al. [6] describe a mechanism to use the GPU to execute iterations speculatively in parallel. The misspeculation check is also performed on the GPU. In case of misspeculation, the incorrectly executed iterations are identified and executed on the CPU if there are other misspeculated iterations depending on them; otherwise they are executed again on the GPU. Our work differs from both these approaches as we do not execute any iterations speculatively on the GPU. Our work can be enhanced by running the loop sequentially on the CPU while the dependence computation is going on, so that in case the number of levels in the loop is too high to benefit from running on the GPU, the GPU run can be stopped and the data from the CPU run can be used. Kim et al. [8] describe a system to speculatively parallelize loops using pro-

filed data and executing the iterations on a cluster. It tries to optimize the communication and validation overheads.

Zhuang et al. [21] describe a system to compute data dependences using multiple cores while a thread is executing the iterations sequentially. The focus of this work is to reduce the overheads of dependence computation. They partition the iterations into as many chunks as the number of available cores such that each core computes dependences among the iterations in its chunk. But the problem with this approach is that the iterations in chunk *i+1* are assumed to be dependent on the iterations in chunk *i*. Our approach does not have this drawback as it can compute dependences among all the iterations and so can find more parallelism.

The work done by Zhang et al. [19] handles dynamic irregularities in both control flows and memory accesses. They use the concepts of data reordering and job swapping to remove the dynamic irregularities. These techniques can be used to improve the performance of any GPU kernel. Lee et al. [9] have developed a compiler framework to do automatic source-to-source translation of OpenMP applications to CUDA applications. They discuss various transformations to reduce the cost of GPU global memory accesses. But their work primarily considers data parallel loops.

Hardware Transactional Memory for GPUs has been proposed by Fung et al. [7]. The design uses word-level, value-based conflict detection. Also it uses speculative validation using a bloom filter based approach to improve transaction commit parallelism.

A scalable approach to dynamic data dependence profiling has been discussed by Kim et al. [11]. It focuses on reducing the memory overheads by stride detection and compression, and reducing the runtime overhead by parallelizing the data-dependence profiling process on multiple cores. Another approach to reduce the memory requirements and improve the performance, has been discussed by Yu et al. [18]. It partitions the profiling task into multiple independent slices using compiler and runtime techniques. These slices can be profiled in parallel. Data generated for each slice is then combined automatically by the compiler to obtain the complete data dependence graph.

6. Conclusion and Future Work

In this paper, we presented an algorithm to execute loops with cross iteration dependences due to indirect memory accesses, on GPUs. We discussed how the compute capabilities of a GPU can be used to compute memory dependences at runtime. We also discussed how we can levelize the iterations of a loop and speed up the execution of loops by executing iterations at a level in parallel. We also described how the dependence computation and actual computation can be pipelined. This mechanism can be used to speed up loops with a reasonable number of cross iteration dependences.

Current work assumes that the arrays with indirect accesses are non-overlapping. In future we would like to extend this technique to handle overlapping arrays. We would also like to work on reducing the kernel call overheads. In addition to these, we would like to handle the cases where the GPU memory is not sufficient to hold the kernel data in addition to the dependence computation data.

7. Acknowledgements

We thank the anonymous reviewers for their suggestions and comments. We also thank Sreepathi Pai and other members of the Lab for HPC for discussions and feedback on improving the paper. The first author acknowledges the funding received from Google India Private Limited.

References

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

[2] M. Baskaran, J. Ramanujam, P. Sadayappan. *Automatic C-to-CUDA code generation for affine programs*. In CC, 2010.

[3] B. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, B. Pottenger, L. Rauchwerger, P. Tu, S. Weatherford. *Polaris: The Next Generation in Parallelizing Compilers*. In LCPC, 1994.

[4] D. K. Chen, P. C. Yew, J. Torellas. *An efficient algorithm for the run time parallelization of doacross loops*. In Supercomputing, 1994.

[5] J. Saltz, R. Mirchandaney, K. Crowley. *Run-time parallelization and scheduling of loops*. IEEE Trans. Computers, 1991.

[6] M. Feng, R. Gupta, L. N. Bhuyan. *Speculative Parallelization on GPGPUs*. In PPoPP, 2012.

[7] W W. L. Fung, I. Singh, A. Brownsword, T M. Aamodt *Hardware Transactional Memory for GPU Architectures*. In MICRO, 2011.

[8] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, D. I. August. *Automatic Speculative DOALL for Clusters*. In CGO, 2012.

[9] S. Lee, S. J. Min. R. Eigenmann. *OpenMP to GPGPU: a compiler framework for automatic translation and optimization*. In PPoPP, 2009.

[10] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym. *NVIDIA Tesla: A Unified Graphics and Computing Architecture*. IEEE Micro, 2008.

[11] M. Kim, H. Kim, C Luk. *SD3: A Scalable Approach to Dynamic Data-Dependence Profiling*. In MICRO, 2010.

[12] L. N. Pouchet. *The Polyhendral Benchmark suite*. http://www.cse.ohio-state.edu/~pouchet/software/polybench.

[13] NVIDIA Corp, *NVIDIA CUDA: Compute Unified Device Architecture: Programming Guide*, Version 4.2, 2012.

[14] NVIDIA Corp, Fermi Compute Architecture White Paper.

[15] L. Rauchwerger, N. Amato, D. Padua. *A scalable method to runtime loop parallelism*. In IJPP, July 1995.

[16] M. Samadi, A. Horamati, J. Lee, S. Mahlke. *Paragon: Collaborative Speculative Loop Execution on GPU and CPU*. GPGPU-5 2012.

[17] Stanford Compiler Group. *SUIF: A parallelizing and optimizing research compiler*. Technical Report CSL-TR-94-620, Stanford University, Computer Systems Laboratory, 1994.

[18] H. Yu, Z. Li, *Multi-slicing: A Compiler-Supported Parallel Approach to Data Dependence Profiling*, In ISSTA, 2012.

[19] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, X. Shen *On-the-Fly Elimination of Dynamic Irregularities for GPU Computing*, In ASPLOS, 2011.

[20] C. Zhu, P. C. Yew. *A scheme to enfore data dependence on large multiprocessor systems*. IEEE Trans. Software Engineering, June 1987.

[21] X. Zhuang, A. E. Eichenberger, Y. Luo, Kevin O'Brien, Kathryn O'Brien. *Exploiting Parallelism with Dependence-Aware Scheduling*, In PACT, 2009.