An Analytical Model-based Auto-tuning Framework for Locality-aware Loop Scheduling

Rengan Xu[†], Sunita Chandrasekaran[‡], Xiaonan Tian[†], and Barbara Chapman[†]

[†]Department of Computer Science, University of Houston, Houston, TX, USA, {rxu6, xtian2, bchapman}@uh.edu
[‡]Department of Computer & Information Sciences, University of Delaware, Newark, DE, USA, schandra@udel.edu

Abstract. HPC developers aim to deliver the very best performance. To do so they constantly think about memory bandwidth, memory hierarchy, locality, floating point performance, power/energy constraints and so on. On the other hand, application scientists aim to write performance portable code while exploiting the rich feature set of the hardware. By providing adequate hints to the compilers in the form of directives appropriate executable code is generated. There are tremendous benefits from using directive-based programming. However, applications are also becoming more and more complex and we need sophisticated tools such as auto-tuning to better explore the optimization space. In applications, loops typically form a major and time-consuming portion of the code. Scheduling these loops involves mapping from the loop iteration space to the underlying platform - for example GPU threads. The user tries different scheduling techniques until the best one is identified. However, this process can be quite tedious and time consuming especially when it is a relatively large application, as the user needs to record the performance of every schedule's run. This paper aims to offer a better solution by proposing an auto-tuning framework that adopts an analytical model guiding the compiler and the runtime to choose an appropriate schedule for the loops, automatically and determining the launch configuration for each of the loop schedules. Our experiments show that the predicted loop schedule by our framework achieves the speedup of 1.29x on an average against the default loop schedule chosen by the compiler.

1 Introduction

Heterogeneous architectures that comprise of CPU processors and computational accelerators such as GPUs have been increasingly adopted for scientific computing. The low-level programming models CUDA and OpenCL for GPUs offer users programming interfaces with execution models closely matching that of GPU architectures. Effectively using these interfaces for creating highly optimized applications require programmers to thoroughly understand the underlying architecture, as well as significantly change the program structures and algorithms. This affects both productivity and performance. Standardized directive-based

models such as OpenACC [3] and OpenMP for accelerators [5] require developers to insert directives and runtime calls into the existing source code offloading portions of Fortran or C/C++ codes to be executed on accelerators.

Directives are high-level language constructs that programmers can use to provide useful hints to compilers to perform certain transformations and optimizations on the annotated code region. The use of directives can significantly improve programming productivity. Users can still achieve high performance of their program comparable to code written in CUDA or OpenCL, subjected to the requirements that a 'careful' choice of directives and compiler optimization strategies be made. One such scenario encountered quite commonly in a program is loop scheduling.

Loop scheduling defines the mapping of loop nest(s) to the underlying architecture. Consider the architecture is a GPU that consists of a number of GPU threads with complex topology settings; it is a daunting task to determine mapping strategies of the loop nest to those threads in order to achieve better performance. Different loop schedules reflect different memory access orders. Loop transformations change memory access orders that results in exploiting better locality - temporal and spatial. The reuse distance model [4] is a classic model predominantly used for CPUs to capture both types of locality in CPU architectures. However we cannot apply this model directly on GPUs due to the significant architectural differences between the CPUs and the GPUs. This paper aims to extend the reuse model to suit its applicability for GPUs.

The main contributions of this paper include:

- To the best of our knowledge, we are the first to propose a locality-aware auto-tuning framework to address the GPU loop scheduling issue.
- In the proposed framework, we extend the classic reuse distance model to GPU architecture in order to estimate GPU cache hit rate accurately.
- Our results demonstrate that our proposed framework chooses the loop schedule producing better performance compared to the default loop schedule chosen by default by the compiler.

The organization of this paper is as follows: Section 2 gives an overview of GPU architecture and OpenACC model. Section 3 provides a motivating example illustrating the performance impact of different loop schedules. In Section 4, we explain in detail the proposed auto-tuning framework on how to choose a loop schedule with better locality. Performance results are discussed in Section 5. Section 6 highlights the related work in this area. We conclude our work in Section 7.

2 GPU Architecture and OpenACC Directives

GPU architectures differ significantly from that of traditional processors. Employing a Single Instruction Multiple Threads (SIMT) architecture, NVIDIA GPUs have hundreds of cores that can process thousands of software threads simultaneously. GPUs organize both hardware cores and software threads into

two-level of parallelism. Hardware cores are organized into an array of Streaming Multiprocessors (SMs), each SM consisting of a number of cores named as Scalar Processors (SPs). Each SM has its own L1 cache which is not cache coherent, and all SMs share an unified L2 cache.

The compute-intensive part of an application, called kernel, is offloaded to GPUs for parallel execution. The GPU launches massive threads to execute that kernel. The thread unit in GPU scheduling is called a warp (a warp size has 32 threads for NVIDIA GPUs). Multiple warps form a thread block and multiple thread blocks form a grid. Both the thread block and grid can be 1D, 2D, or 3D. For programmers, the challenge to efficiently utilize the massive parallel capabilities of GPUs is to map the kernels to the thread hierarchy, and efficient data layout in the GPU memory hierarchy to maximize coalesced memory access for the threads.

Directive-based high-level programming models for accelerators, e.g. OpenACC and OpenMP extensions for accelerators, have been designed to address the programmability challenge of GPUs. Using these programming models, programmers insert compiler directives into a program to annotate portions of code to be offloaded onto accelerators for executions. This approach relies heavily on the compiler to generate efficient code for thread mapping and data layout. It could be potentially challenging to extract the optimal performance using such an approach rather than using other explicit programming models. However, the directive-based models simplify programming on heterogeneous systems thus saving development time, while also preserving the original code structure assisting in code portability.

OpenACC allows users to specify three levels of parallelism in a data parallel region: gang, worker and vector parallelism to map the loop nests to the multiple-level thread hierarchy of GPUs. Programmer provides hints to map these three-level parallelism to GPU threads but the effectiveness of the mapping relies on the compiler and runtime implementation strategies. We use a high quality, open-source, validated OpenACC compiler called OpenUH [18]. Adhering to OpenACC standards, this compiler maps "gang" to thread block, "worker" to Y-dimension of thread block and "vector" to X-dimension of thread block [3].

3 A Motivating Example

Matrix multiplication has been widely used in scientific computing. We use this application to illustrate the importance of loop scheduling in GPU. The square-matrix multiplication we used is $C = AB$ where the size of matrix A, B, and C is $n \times n$. The elements in matrix C are $C_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$ where both the indices i and j loops from 1 through n. A double nested loop was constructed to solve this matrix multiplication. Multiple ways could be adopted to map this loop nest to the underlying GPU threads using directive-based programming model. Table 1 shows how this loop nest could be mapped in so many different ways to GPU threads. The table also indicates different launch configurations. The launch configuration specifies the thread block and grid shape and size that are used to

Table 1: The performance difference for matrix multiplication with different loop schedules and launch configurations

Loop schedule number	Loop schedule detail	Performance(ms)
0	bx(1)/tx(128)	3.24
1	by(1)/bx(1) tx(128)	3.36
2	by(1) ty(128)/bx(1) tx(1)	11.04
3	by(1) ty(64)/bx(1) tx(2)	5.87
4	by(1) ty(32)/bx(1) tx(4)	4.28
5	by(1) ty(16)/bx(1) tx(8)	3.47
6	by(1) ty(8)/bx(1) tx(16)	3.09
7	by(1) ty(4)/bx(1) tx(32)	3.16
8	by(1) ty(2)/bx(1) tx(64)	3.19
9	by(1) ty(1)/bx(1) tx(128)	3.28

run a loop. All of the 10 loop schedules in the table use only one thread block with 128 threads. However, since the loop nest is mapped to threads differently, there are differences in their performance. The loop schedule 0 is the default loop schedule chosen by the compiler. However, we notice that there are other loop schedules demonstrating better performance than the default schedule that the compiler chose. What are the strategies to choose an optimal or sub-optimal loop schedule and its corresponding launch configuration? Our proposed auto-tuning framework discussed in the rest of the paper provides suitable proven answers to this question.

We would like to keep the notations used for our framework as general as possible and not tie it to any specific programming model/language:

- bx, by and bz: denote X, Y and Z dimension of the grid, respectively
- tx, ty and tz: denote X, Y and Z dimension of the thread block, respectively
- num_bx, num_by and num_bz: denote the size of X, Y and Z dimension of the grid, respectively
- num_tx, num_ty and num_tz: denote the size of X, Y and Z dimension of the thread block, respectively

4 Auto-tuning for GPU Loop Scheduling

4.1 The Auto-tuning Framework

In this section, we describe our auto-tuning framework and the analytical model proposed that enables the identification of the appropriate loop schedule, and the launch configuration used for each of the loop schedules. Figure 1 gives an overview of the auto-tuning framework. The compiler generates multiple kernel files with different loop schedules. The loop schedule is chosen from a set of loop schedule patterns which covers both double and triple nested loops. The framework chooses a launch configuration from the launch configuration search space which depends on the iteration space of each loop. The search of launch

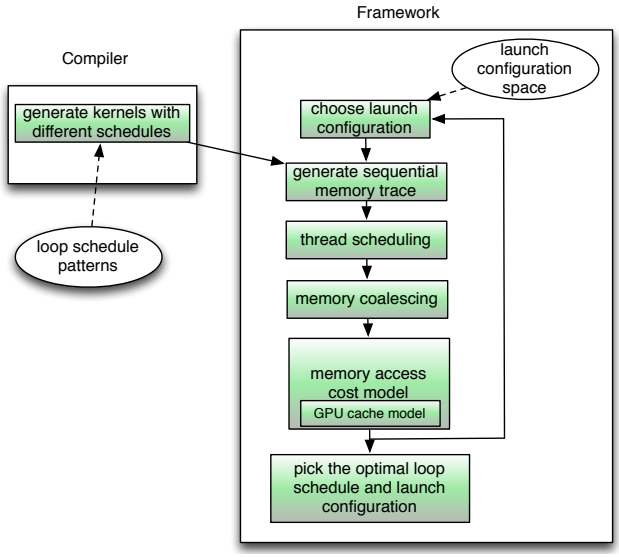


Fig. 1: The framework of auto-tuning for loop scheduling

configuration is guided by the rule of maximizing the GPU occupancy. Before applying the framework, an application needs to run once on CPU to generate a sequential memory trace. Based on the loop schedule and the launch configuration, all memory accesses in the memory trace are assigned to GPU threads. This step defines the access of memory references for GPU threads. Next, the thread scheduling defines the execution order of the memory accesses in the trace.

In the execution of memory accesses, the memory coalescing is very critical. In GPU, a warp (the smallest execution unit) defines a set of consecutive threads. If consecutive threads access consecutive memory addresses, then the memory accesses are coalesced meaning they are merged into fewer memory transactions. We simulate the memory coalescing behavior in GPU architecture in our model. For instance, if the memory addresses referenced by all threads in a warp are in one cache line, then these memory access requests will be merged into only one memory request.

After memory requests are coalesced, the memory trace is fed into the memory access cost model where a memory access cost is computed, with the cache model. This process is repeated until the framework iterates over all loop schedules and the launch configuration space. Finally, the framework picks the optimal loop schedule and the corresponding launch configuration that has the minimal memory access cost. The compiler then recompiles the same program using the selected loop schedule. The major components in this framework will be discussed in the following sub-sections:

4.2 Loop Schedule Patterns

We only consider double and triple nested loops. Note that here the loop nest level means parallelizable loop nest. For instance, in the body of the parallelizable loop nest, there could be another nested loop that is sequentially executed. In the current GPU programming models such as OpenACC, the maximum level of the parallelizable loop nest is three. If a nested loop has more levels to parallelize, it can be collapsed into double or triple nested loop.

Listing 1.1 shows a double nested loop example. Using the notations described in the previous section 3, we now introduce three loop schedules for the double nested loop (use x-loop for the inner loop and y-loop for the outer loop):

- schedule 2_1: x-loop is mapped to the X-dimension of a thread block , and y-loop is mapped to the X-dimension of the grid.
- schedule 2_2: x-loop is mapped to X-dimension of both thread block and grid, and y-loop is mapped to Y-dimension of the grid.
- schedule 2_3: x-loop is mapped to X-dimension of both thread block and grid, and y-loop is mapped to Y-dimension of both thread block and grid.

These loop schedule directives are implicitly added by the compiler. The graphical explanation for these loop schedules are shown in Figure 2, 3 and 4. The detailed mapping function from the loop iterations to GPU threads for double nested loop are shown in Listing 1.3, 1.4 and 1.5. The purpose of schedule 2_2 is to overcome the GPU hardware threads limit within a block. In both schedule 2_1 and 2_2, the threads computing the outer loop are in different thread blocks, which are likely to be scheduled to different GPU SMs (Streaming Multiprocessor). This may not exploit the data locality efficiently. So how do we improve data locality? We consider the loop schedule 2_3 that allows some threads computing the outer loop iterations to remain in the same block thus improving data locality. For triple nested loop, a code example is shown in Listing 1.2 and other similar loop schedules are designed. Because of the space limit, we only illustrate the graphical representation for one loop schedule in Figure 5, in which x-loop, y-loop and z-loop refer to the inner most loop, the middle loop and the outer most loop, respectively. The loop schedule in Figure 5 means x-loop is mapped to X-dimension of thread block, y-loop is mapped to Y-dimension of thread block and z-loop is mapped to X-dimension of the grid.

```
#pragma acc loop
for(j = j_start; j < j_end; j++){
    #pragma acc loop
    for(i = i_start; i < i_end; i++){
        ......
    }
}
```

Listing 1.1: Double nested loop example

```
#pragma acc loop
for(k = k_start; k < k_end; k++){
  #pragma acc loop
  for(j = j_start; j < j_end; j++){
    #pragma acc loop
    for(i = i_start; i < i_end; i++){
            ......
    }
  }
}
```

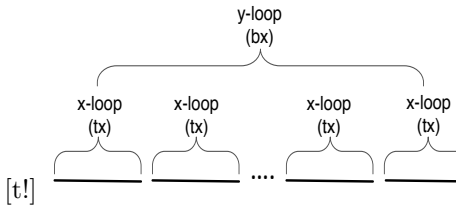
Listing 1.2: Triple nested loop example

```
#pragma acc loop bx(num_bx)
for(j = j_start; j < j_end; j++){
    #pragma acc loop tx(num_tx)
    for(i = i_start; i < i_end; i++){
        ......
    }
}
mapping function to CUDA:
j = j_start + blockIdx.x + t * gridDim.x, (t = 0, 1, ..., (j_end - j_start)/(gridDim.x) - 1)
i = i_start + threadIdx.x + t * blockDim.x, (t = 0, 1, ..., (i_end - i_start)/(blockDim.x) - 1)
```

Listing 1.3: Loop schedule 2_1

```
#pragma acc loop by(num_by)
for(j = j_start; j < j_end; j++){
    #pragma acc loop bx(num_bx) tx(num_tx)
    for(i = i_start; i < i_end; i++){
        ......
    }
}
mapping function to CUDA:
j = j_start + blockIdx.y + t * gridDim.y
(t = 0, 1, ..., (j_end - j_start)/(gridDim.y) - 1)
i = i_start + threadIdx.x + blockIdx.x * blockDim.x + t * blockDim.x * gridDim.x
(t = 0, 1, ..., (i_end - i_start)/(blockDim.x * gridDim.x) - 1)
```

Listing 1.4: Loop schedule 2_2



[t!]

Fig. 2: Loop schedule 2_1

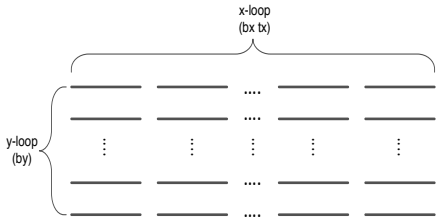
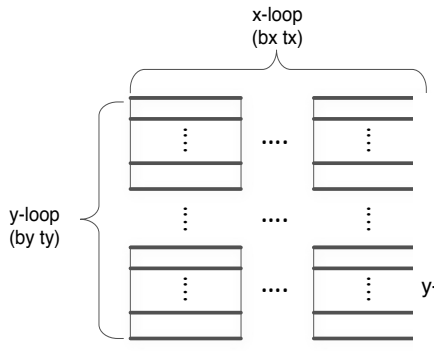


Fig. 3: Loop schedule 2_2



[t!]

Fig. 4: Loop schedule 2_3

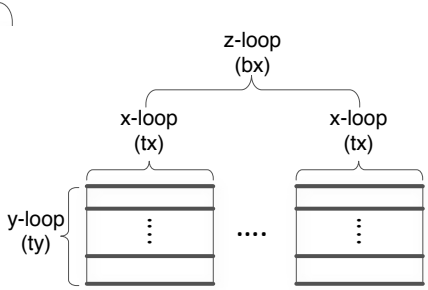


Fig. 5: Loop schedule 3_1

```
#pragma acc loop by(num_by) ty(num_ty)
for(j = j_start; j < j_end; j++){
```

```
    #pragma acc loop bx(num_bx) tx(num_tx)
    for(i = i_start; i < i_end; i++){
        ......
    }
}
mapping function to CUDA:
j = j_start + threadIdx.y + blockIdx.y * blockDim.y + t * blockDim.y * gridDim.y
(t = 0, 1, ..., (j_end - j_start)/(blockDim.y * gridDim.y) - 1)
i = i_start + threadIdx.x + blockIdx.x * blockDim.x + t * blockDim.x * gridDim.x
(t = 0, 1, ..., (i_end - i_start)/(blockDim.x * gridDim.x) - 1)
```

Listing 1.5: Loop schedule 2_3

4.3 Thread Scheduling

The memory trace is defined for how the memory is accessed by threads, which is further defined for how the thread blocks are scheduled into different SMs and how the threads are scheduled within each SM. When the GPU launches a grid of threads for a kernel, that grid is divided into 'waves' of thread blocks. For example let us assume there are 15 SMs. Each SM has 2 thread blocks hence 30 thread blocks in total. Thread block 0 and thread block 15 will be assigned to SM0. Thread block 1 and thread block 16 will be assigned to SM1. If there was a scenario with 60 thread blocks and each SM allows at most 2 blocks (30 blocks for 15 SMs), we will need to assign these blocks into two waves; 30 thread blocks to the first wave and the other 30 thread blocks to the second wave. We use round-robin scheduling mechanism to schedule the thread blocks to all SMs in all waves. The number of threads scheduled is independent of the grid size. For instance, if the grid size is 2048 and only 128 threads are scheduled, then each thread will process 2048/128=16 elements.

Figure 6 shows thread scheduling mechanism. It highlights two waves scheduled to one SM. Each wave has two thread blocks; each thread block has two warps; each warp has two threads; each thread has five memory accesses. We access the memory references in a round-robin manner. This memory access pattern gives us the memory trace.

The equation to calculate the number of waves is given in Equation 1. The number of waves is obtained by dividing the total number of thread blocks by the active thread blocks per SM times the number of SMs. The active blocks per SM is given in Equation 2. For instance, in Kepler GPU, the $max\_threads\_per\_SM$ is 2048 and $max\_thread\_blocks\_per\_SM$ is 16 and upon knowing the number of thread blocks in the kernel, which is specified by the launch configuration, we can determine the number of waves.

$$waves = \frac{thread\_blocks}{active\_blocks\_per\_SM \times \#SMs} \tag{1}$$

$$active\_blocks\_per\_SM = min(max\_threads\_per\_SM/block\_size, max\_thread\_blocks\_per\_SM) \tag{2}$$

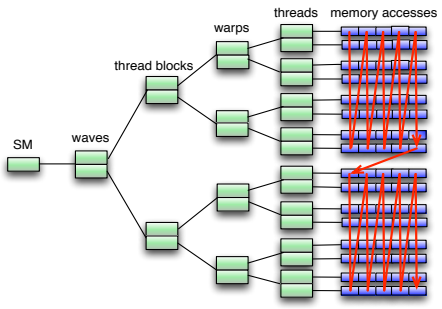


Fig. 6: Thread scheduling used in the auto-tuning framework

4.4 Memory Access Cost Model

After memory coalescing, the memory trace is fed into the memory access cost model which computes the memory access cost for a specific loop schedule and launch configuration. The metric used in this model is presented as

$$Cost_{mem} = \sum_{i}^{\#levels} (N_i \times L_i) \tag{3}$$

where N_i means the number of transactions happened in level i of the memory hierarchy, and L_i means the latency of memory level i.

The rationale behind this metric is the memory hierarchy in GPU architecture which is shown in Figure 7. When the kernel needs to access a global memory address, it needs to load that address from L1 cache. If the data is already in L1 cache, then the access is a hit. If the data is not in L1 cache, then the access is a miss and it needs to load the data from L2 cache. If the data is not in L2 cache, then it needs to further load the data from DRAM. So the formula after expanding the Equation 3 is shown in Equation 4, which is the sum of memory access cost from L1, L2 and DRAM. The formula for calculating each individual cost is given in Equation 5, Equation 6 and Equation 7. The '4' in Equation 6 and Equation 7 explains the number of global memory load transaction that is increased by 1 every 128 bytes in L1 cache, but every 32 bytes in L2 cache and DRAM. Since the memory access latency order from high to low - DRAM, L2 cache and L1 cache, we would like to access higher order memory as less as possible. In other words, we would like to have few global loads and low L1 and L2 cache miss rates as possible. When there is intra-thread data reuse or inter-thread data reuse, different loop schedules have different cache miss rates, and finally the performance of the kernels using those loop schedules would be different.

$$Cost_{mem} = Mem_{L1} + Mem_{L2} + Mem_{DRAM} \tag{4}$$

$$Mem_{L1} = global\_loads * (1 - L1\_miss\_rate) * L1\_latency \tag{5}$$

$$Mem_{L2} = global\_loads * L1\_miss\_rate * 4 * L2\_latency \tag{6}$$

$$Mem_{DRAM} = global\_loads * L1\_miss\_rate * L2\_miss\_rate * 4 * DRAM\_latency \tag{7}$$

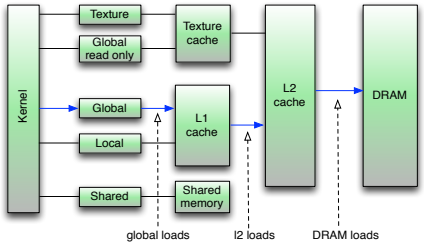


Fig. 7: GPU memory hierarchy

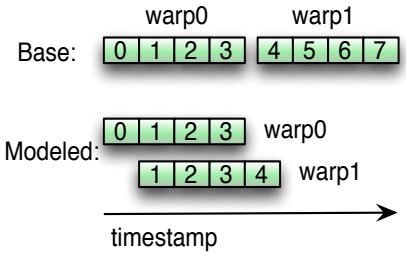


Fig. 8: L1 cache modeling

The key factors of the model are to estimate the global memory loads, L1 and L2 cache miss rates. To estimate L1 and L2 cache miss rate, we use reuse distance model [4]. It is a classic model to model the cache misses in CPU applications. The primary reasons for cache misses are cold/compulsory, conflict, capacity misses, famously termed as the 3C model. The **cold miss** occurs when there is no data in the cache, no matter how big the cache is. The **conflict miss** usually occurs in direct-mapped caches and set-associative caches. Two cache lines may map to the same cache slot even though there may be empty slots. The **capacity miss** happens when there are no more available slots in the cache. The reuse distance model assumes that a LRU replacement fully associative cache is used. So it can only predict cold miss and capacity miss.

To the best of our knowledge, there is no existing work that discusses GPU L2 cache modeling. We found a couple of other related research work discussing GPU L1 cache modeling. Tang et al. [17] applied the reuse distance theory to model the GPU L1 cache. However, there were a few weaknesses and limitations in their approach: (1) they assumed only one thread block is active in one SM which is not true in the real hardware; (2) they modeled the cold miss and conflict miss but did not model capacity miss, however some research have shown that only a minority of the misses are conflict misses in both CPU [6] and GPU [14]; (3) they validated their model against a GPU simulator which is not a real hardware per se. Nugteren et al. [14] also used the reuse distance to model GPU L1 cache. However, in their implementation, all thread blocks were scheduled into only one SM which is not the case in a real hardware. Our thread scheduling mechanism overcomes the drawbacks of the above two papers discussed.

The reuse distance theory can measure both spatial locality and temporal locality if the distance is measured with cache line granularity. The spatial locality defines that the nearby memory addresses are likely to be referenced again

in the near future. The temporal locality defines that the same data is likely to be referenced again in the near future.

The spatial locality is reflected by the memory coalescing level in the GPU kernel. If a GPU kernel has coalesced memory accesses, then it has better spatial locality than the kernel that has uncoalesced memory accesses. This is because the coalesced memory accesses allow the nearby data elements to be accessed at the same time the current data is accessed.

The temporal locality is reflected by the loop schedule. Different loop schedules pose different temporal locality since the execution order of the threads are different. The reuse distance theory can capture both the spatial locality and temporal locality effectively. Table 2 shows a reuse distance example. In this example, assume the cache line is of 16 bytes. If the data is accessed first or when a cold miss happens, the reuse distance is recorded as ∞. The reuse distance is a metric that defines the *distinct* memory accesses between the current memory access and the last access. If the reuse distance is larger or equal to the total number of cache lines, then a data reference is missed in the cache. The cache hit rate can be obtained by diving the hits by the total number of hits and misses.

Table 2: Reuse distance example. Assume cache line has 16 bytes and the cache size is 32 bytes. The reuse distance is based on cache line granularity

address	0	8	16	96	8	16	17	104
cache line	0	0	1	6	0	1	1	6
reuse distance	∞	0	∞	∞	2	2	0	2
cache hit/miss	miss	hit	miss	miss	miss	miss	hit	miss

Although the classic reuse distance model can predict the cache miss rate in CPU, it cannot be simply applied as-is on the GPU since the architectures are significantly different. The most important difference is that in GPU, the threads in a warp execute in lock-step manner and therefore memory coalescing is important in the memory accesses of a warp. If the memory addresses referenced by all the threads in a warp are in a cache line, then the memory accesses are merged into one memory access. Another difference is the parallel processing feature including parallel memory processing in GPU. Therefore in our implementation, the L1 modeling includes parallel memory processing. But we also compare it with the base implementation. The difference of "Base" and "Modeled" are shown in Figure 8. In Base version, the memory coalescing is applied to the memory trace. Then the memory requests from different warps are processed in order. If the memory requests in a warp are not coalesced, then they are also processed in order within a warp. In Modeled version, we also apply memory coalescing, but we further add a timestamp. The timestamp is added to the following warps but it is also added to the threads in the same warp if their memory requests are not coalesced.

In the reuse distance model implementation, a key factor is the input, which is a memory trace. In our analytical model, the memory traces are different for different loop schedules. This is because different loop schedules assign the loop iterations into GPU threads differently, therefore the memory traces are different, and eventually the cache misses are different.

For L2 cache modeling, we must first apply L1 cache modeling for all SMs and record the cache misses in their individual list. Then the memory trace is processed in round-robin manner which is similar to the description in Figure 6.

5 Performance Evaluation

The experimental platform is Intel Xeon processor E5520 with frequency 2.27GHz and 32GB main memory and an Nvidia Quadro K6000 GPU card which uses K40 architecture. L1 and L2 cache sizes are 16KB and 1.5MB, respectively. The cache line size for both L1 and L2 is 128 bytes. The proposed framework is implemented within the OpenUH compiler. The actual L1 and L2 cache hit rates are obtained from l1_cache_global_hit_rate and l2_l1_read_hit_rate metrics in CUDA profiler nvprof and the actual global memory loads are obtained from gld_transactions metric.

To evaluate our auto-tuning framework, we consider several benchmarks: two synthetic benchmarks (x-reuse and y-reuse), four from kernelGen OpenACC Performance Test Suite [2] (Matrix Multiplication, Jacobi, Laplacian and Divergence), one from CUDA SDK (Matrix Transpose) and one from EPCC OpenACC benchmarks [1] (Himeno). We test different data reuse patterns using the two synthetic benchmarks. Figure 9 shows these two benchmarks along with another pattern i.e. xy-reuse, a classic Matrix Multiplication case. The "x" here refers to the inner loop and "y" refers to the outer loop in a double nested loop. In the x-reuse benchmark, the inner loop reuses the common data; while in the y-reuse benchmark, the outer loop reuses the common data. The third case is the xy-reuse where both the inner and the outer loop reuse some common data.

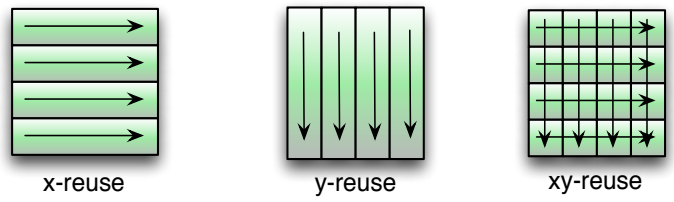


Fig. 9: Data reuse patterns

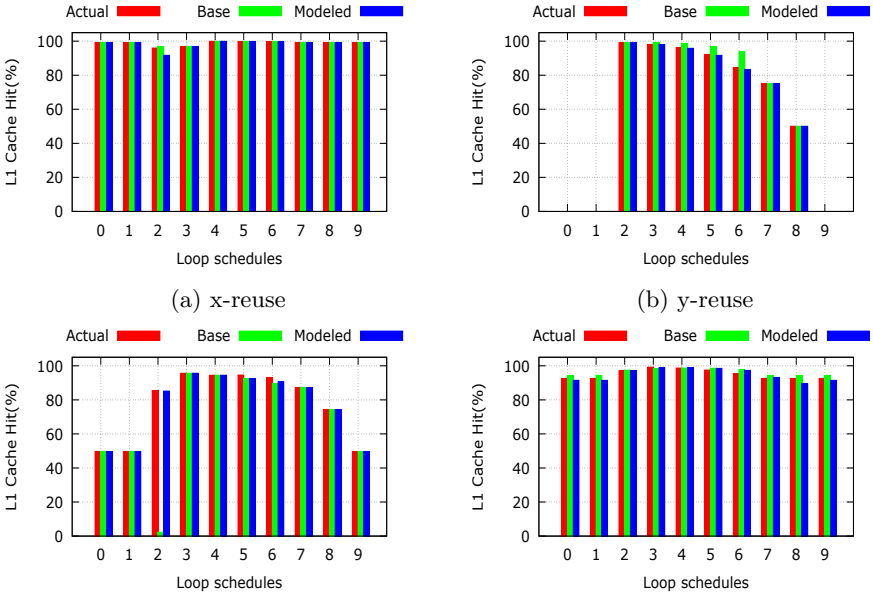
Figure 10 shows results for L1 cache hit modeling for some of the benchmarks discussed above. Figure (a) and (b) are results for the two synthetic benchmarks and Fig (c) and (d) are results for a couple of benchmarks from kernelGen suite.

(Results for other benchmarks were quite similar, so due to space constraints we have not included them in the paper). The results indicate that modeled result is more accurate than "Base" version since it considers the parallel memory processing. Figure 10(a) shows that cache hit rates are high for all loop schedules. This is because for all iterations in x loop, the data they share are in one row and in the same contiguous memory section. Figure 10(b) shows that the shared data are in the same column and therefore they are not contiguous in memory. This leads to relatively lower cache hit. Figure 10(c), results for Matrix Multiplication, show that there is data reuse in both x and y loops and therefore the shape of cache hit results seem like a combination of x-reuse and y-reuse. Figure 10(d), results for Jacobi show that, the overall hit rate is slightly lesser than x-reuse. This is because the data, the threads share are stencil-like. For instance considering a 4-point stencil, for different points, the data that the threads access are not in contiguous memory locations, however for a specific point the data, the threads share, are still in contiguous memory location. As a result the cache hit rates are still relatively high. If the cache hit is high, the indication is that the threads will take lesser time to fetch data from high-latency memory.

The GPU L2 cache modeling result is shown in Figure 11. We show the results for partial benchmarks including Laplacian, Divergence and Himeno. The results indicate that some loop schedules have low L2 cache hit while other loop schedules have high L2 cache hit. This illustates the importance of choosing the right loop schedules. The error percentage between the actual and the modeled L2 hit is only 4.37%, 13.72% and 2.76% for Laplacian, Divergence and Himeno, respectively. The low error percentages indicate that our model can capture the L2 locality for different loop schedules accurately.

Figure 12 shows the global memory loads of kernels for the four benchmarks discussed in Figure 10. The plots show that the modeled loads (before kernel launch) are exactly the same as the actual loads (profiled results) thus indicating that our proposed model is accurately predicting the memory loads. Figure 12 (b) indicates that for y-reuse synthetic benchmark, no matter what the loop schedule is, the memory access appears to be fully coalesced leading to the same number of global memory loads all the time. In the other three plots, the tallest bars indicate the loop schedules for which the memory accesses are fully uncoalesced, while the shortest bars indicate the loop schedules for which memory accesses are fully coalesced, and the bars between the tallest and the shortest bars indicate partial memory coalescing. (Results for other benchmarks in kernelGen suite and EPCC were quite similar, so due to space constraints we have not included them in the paper). Higher the global memory loads, higher the time taken by the threads to process the memory requests.

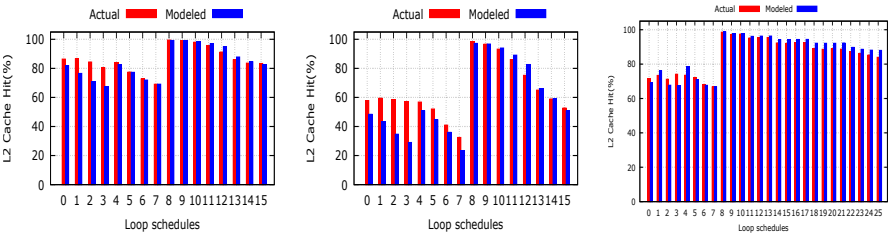
Figure 13 shows several plots that demonstrate close correlation of kernel performance and the memory access cost modeling. We use the coefficient of determination R^2 to measure the strength of the relationship between the kernel performance and the memory access cost in the model. R^2 is a popular indicator on how well a variable can be used to predict the value of another variable. The values of R^2 range from 0 (poor indicator) to 1 (excellent predictor). The R^2



(a) x-reuse (b) y-reuse

(c) Matrix Multiplication (xy-reuse) (d) Jacobi

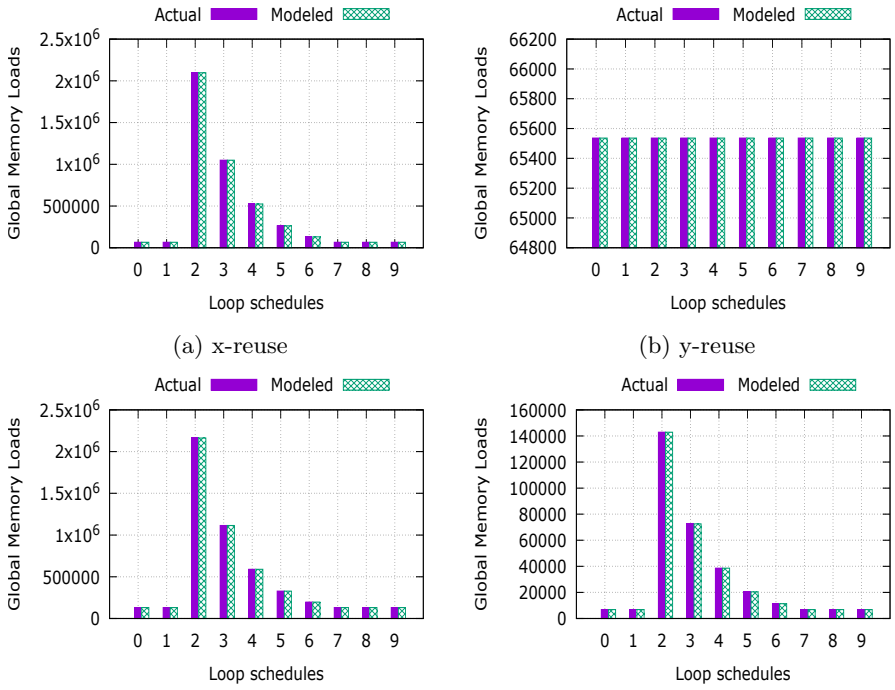
Fig. 10: GPU L1 cache modeling



(a) Laplacian (b) Divergence (c) Himeno

Fig. 11: GPU L2 cache modeling

value for all benchmarks are listed in Table 3 and the average value is 0.93 indicating the strong correlation between the kernel performance and the memory access cost modeling. Based on the memory access cost modeling, an optimal or a sub-optimal loop schedule is chosen by the framework. For all benchmarks tested, the speedup of the loop schedule chosen by the model against the default loop schedule chosen by the compiler are listed in Table 3 and the average speedup is 1.29x. This proves the effectiveness of the proposed framework.



(a) x-reuse

(b) y-reuse

(c) Matrix Multiplication (xy-reuse)

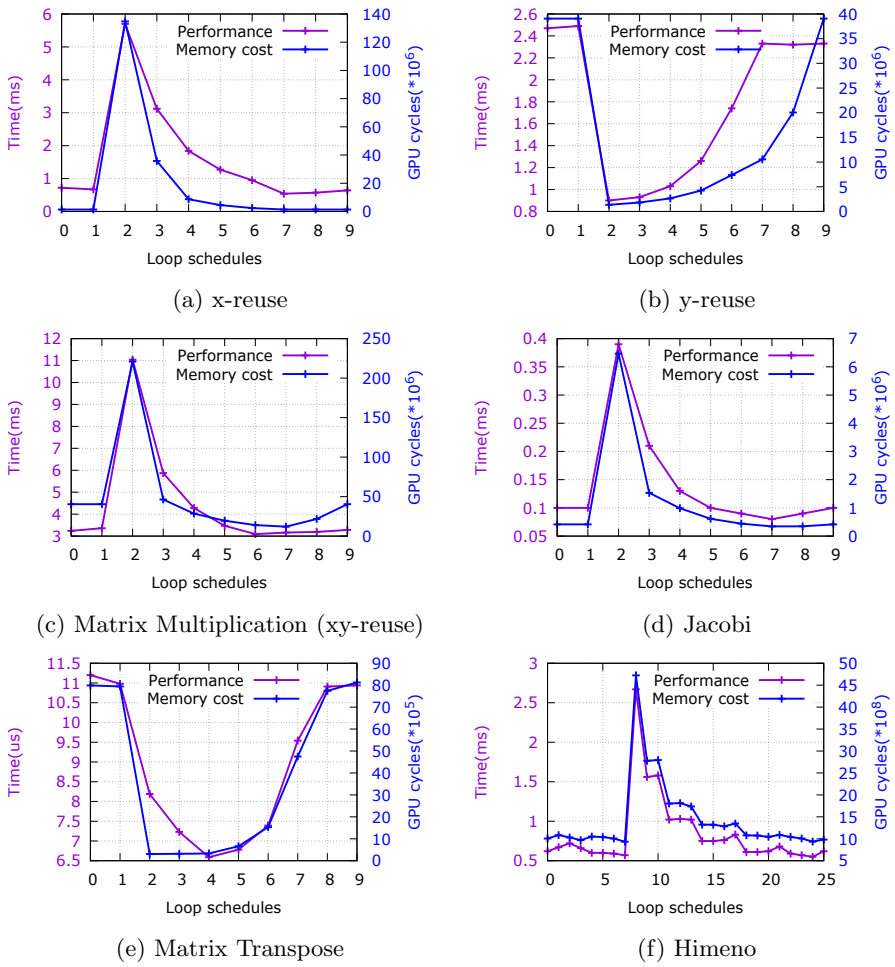
(d) Jacobi

Fig. 12: Global memory loads

6 Related Work

To the best of our knowledge, there were only a few similar research on how to solve the loop scheduling issues on a GPU. Siddiqui et al. [16] presented how to choose the optimal loop schedule with machine learning approach. They used exhaustive search to find the optimal and sub-optimal loop schedules for the training data sets and stored those information into a database. For the new test benchmark, they found the closest training benchmark and applied its loop schedules to that test benchmark. Their approach, however, could only be used for different problem sizes in the same application because it was difficult to define how close two applications are. Instead of exhaustive search, Montgomery et al. [13] used more efficient search approach such as direct search to find the optimal loop schedule. Their approach required to execute the kernels with different loop schedules. Our approach, however, only needs to run the kernel once on CPU because the model predicts the optimal loop schedule before the kernel's execution on GPU. This is one of the major highlights of our proposed framework.

Lee et al. [11] presented a framework to automatically and efficiently map a nested loop to GPU using Domain Specific Language (DSL). The parameters



(a) x-reuse (b) y-reuse

(c) Matrix Multiplication (xy-reuse) (d) Jacobi

(e) Matrix Transpose (f) Himeno

Fig. 13: Plots demonstrating correlation of Performance vs Memory Access Cost Modeling

that form the search space included the dimension of the nested loop, the block size and degree of parallelism, which is essentially the grid size. They applied some hard constraints and soft constraints to restrict the search space. For all loop schedules we consider, many of them have the same software constraints such as the level of memory coalescing. The cache locality, which is a key factor, was not considered in their model.

We looked into some of the other related work on auto-tuning, [9, 8, 12]; these papers obtained the performance of auto-tuning the kernels by running those kernels. Our goal is different from them since we use an analytical model to predict the loop schedule without running the kernel. Hu et al. [10] and

Table 3: Evaluation results

Benchmark	Source	Nested Loop Type	R^2	Speedup
x-reuse	synthetic	double	0.927	1.0
y-reuse	synthetic	double	0.683	2.74
Matrix Multiplication	Performance Test Suite	double	0.913	1.03
Jacobi	Performance Test Suite	double	0.998	1.1
Laplacian	Performance Test Suite	triple	0.999	1.05
Divergence	Performance Test Suite	triple	0.999	0.96
Matrix Transpose	CUDA SDK	double	0.943	1.37
Himeno	EPCC	triple	0.994	1.09

Baghsorkhi et al. [5] used analytical models to predict the performance of a kernel. Our work, however, do not need to predict the execution time of a kernel accurately because the computation part is the basically the same for all loop schedules and the difference of the performance part is the memory cost. Other research include the effect of cache on GPU applications. Picchi et al. [15] applied L2 cache locking mechanism to improve GPU application performance. Choi et al. [7] analyzed the L1 and L2 cache behavior for some benchmarks with a GPU simulator.

7 Conclusion and Future Work

This paper discusses the importance of auto-tuning loop scheduling for GPU computing. We propose an analytical model-based auto-tuning framework to find the optimal or sub-optimal loop schedule that is better than the default loop schedule chosen by the compiler. The model used in the framework is locality-aware as it can predict the cache locality for each loop schedule. The model also predicts the total number of global memory loads and based on these information it obtains a memory access cost for each loop schedule. The framework iterates over all loop schedule patterns and launch configuration space and picks the loop schedule with the least memory access cost. We analyze the proposed framework with multiple benchmarks. The results indicate that the memory access cost modeling has strong correlation with the kernel performance and the loop schedule picked by the framework can achieve 1.29x speedup over the default loop schedule chosen by the compiler. For the future work, we will integrate more factors into the model to improve the prediction of the loop schedule that is as close as the optimal loop schedule.

References

1. EPCC OpenACC Benchmarks. `https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openacc-benchmark-suite`, 2015.

2. KernelGen Performance Test Suite. `https://hpcforge.org/plugins/mediawiki/wiki/kernelgen/index.php/Performance_Test_Suite`, December 2015.
3. OpenACC. `http://www.openacc.org`, 2016.
4. G. Almási, C. Caşcaval, and D. A. Padua. Calculating Stack Distances Efficiently. In *ACM SIGPLAN Notices*, volume 38, pages 37–43. ACM, 2002.
5. S. S. Baghsorkhi, M. Delahaye, W. D. Gropp, and W. H. Wen-mei. Analytical Performance Prediction for Evaluation and Tuning of GPGPU applications. In *Workshop on EPHAM'09, In conjunction with CGO*. Citeseer, 2009.
6. K. Beyls and E. DHollander. Reuse Distance as a Metric for Cache Behavior. In *Proceedings of the IASTED Conference on Parallel and Distributed Computing and systems*, volume 14, pages 350–360, 2001.
7. K. H. Choi and S. W. Kim. Study of Cache Performance on GPGPU. *IEIE Transactions on Smart Processing & Computing*, 4(2):78–82, 2015.
8. X. Cui, Y. Chen, C. Zhang, and H. Mei. Auto-tuning Dense Matrix Multiplication for GPGPU with Cache. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 237–242. IEEE, 2010.
9. S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos. Auto-tuning a High-Level Language Targeted to GPU Codes. In *Innovative Parallel Computing (InPar), 2012*, pages 1–10. IEEE, 2012.
10. Y. Hu, D. M. Koppelman, S. R. Brandt, and F. Löffler. Model-Driven Auto-Tuning of Stencil Computations on GPUs. In *Proceedings of the 2nd International Workshop on High-Performance Stencil Computations*, pages 1–8, 2015.
11. H. Lee, K. J. Brown, A. K. Sujeeth, T. Rompf, and K. Olukotun. Locality-Aware Mapping of Nested Parallel Patterns on GPUs. In *Microarchitecture (MICRO), 47th Annual IEEE/ACM International Symposium on*, pages 63–74. IEEE, 2014.
12. A. Mametjanov, D. Lowell, C.-C. Ma, and B. Norris. Autotuning Stencil-based Computations on GPUs. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 266–274. IEEE, 2012.
13. C. Montgomery, J. L. Overbey, and X. Li. Autotuning OpenACC Work Distribution via Direct Search. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, page 38. ACM, 2015.
14. C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. Bal. A Detailed GPU Cache Model Based on Reuse Distance Theory. In *High Performance Computer Architecture (HPCA)*, pages 37–48. IEEE, 2014.
15. J. Picchi and W. Zhang. Impact of L2 Cache Locking on GPU Performance. In *SoutheastCon 2015*, pages 1–4. IEEE, 2015.
16. S. Siddiqui, F. AlZayer, and S. Feki. Historic Learning Approach for Auto-tuning OpenACC Accelerated Scientific Applications. In *High Performance Computing for Computational Science–VECPAR 2014*, pages 224–235. Springer, 2014.
17. T. Tang, X. Yang, and Y. Lin. Cache Miss Analysis for GPU Programs Based on Stack Distance Profile. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 623–634. IEEE, 2011.
18. X. Tian, R. Xu, Y. Yan, Z. Yun, S. Chandrasekaran, and B. Chapman. Compiling A High-Level Directive-based Programming Model for GPGPUs. In *Intl. workshop on LCPC 2013*, pages 105–120. Springer International Publishing, 2014.