Model-Driven Auto-Tuning of Stencil Computations on GPUs

Yue Hu[1,2], David M. Koppelman[1,2], Steven R. Brandt[1,3], and Frank Löffler[1]

[1]Center for Computation and Technology, Louisiana State University, Baton Rouge, LA, USA

[2]School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, LA, USA

[3]Department of Computer Science, Louisiana State University, Baton Rouge, LA, USA

yhu14@lsu.edu, koppel@ece.lsu.edu, sbrandt@cct.lsu.edu , knarf@cct.lsu.edu

ABSTRACT

Stencil computations are a class of algorithms which perform nearest-neighbor computation, often on a multi-dimensional grid. This type of calculation forms the basis for computer simulations across almost every field of science. The increasing computational speed of graphics processing units (GPUs) make their use for stencil computations an interesting goal. However, achieving highly efficient implementations is often nontrivial, as numerous publications attest. In this work, we propose an analytic performance model for stencil codes on GPUs, which both delivers close-to optimal performance, but at the same time does not require extensive tuning at compile or run time. We evaluate the effectiveness of our performance model using different stencil benchmarks and with various stencil radii.

1. INTRODUCTION

One defining property of stencil computations is that they operate on multi-dimensional arrays (grids), and an update of one particular element of this array (a grid point) only requires information about the nearest-neighbors within a given distance (defining the stencil radius).

Previous works show that graphics processing units (GPUs) are an effective device to accelerate stencil code[1–6]. They also show that an effective and intelligent use of the different kinds of available memory is vital to achieve even moderate performance [2, 3].

In this work, we propose an analytic performance model for stencil code to enable the assignment choice to be made without the need for test runs. We implement three code auto-generation strategies: one without any buffering, one buffered with shared memory, and one buffered with both shared memory and registers. Then we build a performance model that takes thread parallelism, instruction overhead, memory access overhead, and latency hiding ability into consideration. We employ this model to search for the optimal block configuration for each strategy, and predict the best performance. We evaluate the effectiveness of the performance model with four stencil benchmarks and with various stencil radii.

Because our calculation allows us to model the performance accurately, we are able to evaluate performance using a small fraction of the time required to do a test run, and so we are able to consider a larger space than we otherwise would, and to find more opportunities for optimization.

This work is part of Chemora[7] project, which is a code generation and optimization framework. It takes a high-level problem description in terms of partial differential equations and generates highly optimized code suitable for a wide range of heterogeneous systems[8–11]. Chemora separates code generation and optimization into layers, starting from the high level mathematical description and progressing to low level optimizations for specific architectures.

HiStencils 2015
Second International Workshop on High-Performance Stencil Computations
January 20, 2015, Amsterdam, The Netherlands
In conjunction with HiPEAC 2015.

http://www.exastencils.org/histencils/2015/

2. RELATED WORK

Stencil computation is challenging for modern CPUs and GPUs because of the high memory bandwidth requirements. Spatial blocking [1–6] is a commonly used technique to alleviate both latency and bandwidth problem of off-chip global memory. Maruyama and Aoki [3] study spatial blocking with read-only cache and shared memory on NVIDIA's Kepler GPUs. Nguyen et al. [2] present a novel 3.5D-blocking algorithm that performs 2.5D-spatial blocking and 1D-temporal blocking for stencil computations on CPUs and GPUs.

Manually writing and tuning stencil code is time-consuming as well as requiring in-depth knowledge of the CPU and GPU architectures. A number of recent studies[12–16] have focused on stencil code auto-generation and auto-tuning. Holewinski et al. [12] develop compiler algorithms for automatic stencil code generation on GPU accelerators. Lutz et al. [13] propose an auto-tuning framework specifically for stencil computation on multi-GPU systems. Zhang et al. [14] develop an auto-tuning technique for the auto-generated stencil code on GPUs. They first shrink the parameter space to a small set (which could be more than 100 in the worst case), then auto-tune the code by running stencil code with each parameter group and selecting the one with the best performance. Similarly, the auto-tuning environment built by Datta et al. [15] also requires the program to run with hundreds of different parameters and then select the best one.

Meng et al. [16] establish a performance model to automatically select the optimal ghost zone size and generate appropriate stencil code. They consider the effect of computation, memory transfer, and synchronization on the final performance. Since the purpose of their work is to study ghost zone optimization, they only model the strategy we call "buffered with shared memory". In addition, they simplify the model by assuming block configuration is a constant, i.e. the number of threads in a block along each dimension are equal. Hong et al. [17] propose an analytic GPU performance model, but their model are not suitable for auto-tuning of stencil computation as it is built for general GPU programs and it also requires manually counting some metrics. Moreover, their model does not take shared memory bank conflict into consideration.

3. PRELIMINARIES

The auto-tuning system described here works with CUDA by NVIDIA, and we use CUDA terminology. A GPU consists of multiple streaming multiprocessors(*SMs*), 14 for the NVIDIA Kepler K20xm. GPU code executes in a unit called a *kernel*, which consists of threads organized into *blocks*. Threads in the same block can quickly share data. The number of blocks and the number of threads per block make up a *launch configuration*. Threads are scheduled for execution in groups called *warps*, all sharing an instruction. The warp size for the K20xm and all prior NVIDIA products is 32. The rate of execution is limited by *warp issue throughput* and *dependency stalls*, the later of which has many causes. Warp issue throughput varies by device and instruction. This work assumes a few values. Double-precision arithmetic operations issue at 2 warps per cycle per SM ($\theta_{\mathrm{dp}} = 2\,\mathrm{wp/cyc}$). A block consisting of B threads, each of which executes N DP instructions would take $B\frac{N}{32\theta_{\mathrm{dp}}}$ cycles to issue on an SM.

Dependency stalls occur when there is no warp whose source operands are ready. The length of such stalls is determined by *latency*, *congestion*, and *dependence distance* effects. Many non-memory instructions have an 11-cycle latency, which we assume is *hidden*, meaning that dependent instructions can always be scheduled 11 cycles later avoiding any stall. Global memory latency (the time between a load instruction and the arrival of uncached data), $T_{\mathrm{gmem\_lat}}$, is much larger, 200-400 cycles[18]; the model uses $T_{\mathrm{gmem\_lat}} = 200\,\mathrm{cyc}$ for the K20xm.

Execution of a load instruction entails constructing memory read *requests*, each requesting 32, 64, or 128 contiguous bytes. Congestion refers to the additional time added to read requests due to off-chip bandwidth limitations. For load instructions congestion is due to load instructions requesting data faster than it can be retrieved. For modeling congestion it is convenient to work with the rate at which off-chip data can be retrieved per SM in units of double-precision elements. For the K20xm,

$$\theta_{\mathrm{gmem/sm}} = \frac{250\,\mathrm{GiB/s}}{732\,\mathrm{MHz} \times 14 \times 8\,\mathrm{B/elem}} = 3.7242\,\mathrm{elem/cyc}.$$

4. OVERVIEW

A Chemora code is compiled with a user-written stencil kernel which uses grid data functions, distributed regular meshes. At runtime the auto-tuning system will choose a configuration, taking into account local grid data size, stencil shape, GPU capabilities, etc. It does this by sampling a large space of configurations, and estimating the performance of each one. The kernel code will be compiled (at runtime) using the best configuration.

A configuration consists of the CUDA launch configuration, an iteration direction and count, and a buffering strategy. The buffering strategy is the method used to access the array. The strategies are discussed below. The model can be used to predict the best performance given a buffering strategy.

When performing stencil computations on a GPU one must reconcile several competing goals. On the one hand we want to minimize thread block surface area to maximize re-use, on the other hand we want to make efficient use of the memory system which means fully using requests and avoiding underutilization due to the thread block dimensions not being a multiple of the respective local data size.

In addition, shared memory bank conflicts should be minimized. For these reasons and more, a configuration cannot be chosen by a simple closed form calculation. The model will consider these factors and more to come up with a prediction of execution time.

The performance can be affected by any one of a number of factors. Those considered by the model are: instruction throughput, off-chip memory bandwidth, exposed latencies, and contentions of various types.

Off-chip memory utilization depends heavily on the buffering strategy used, and so we turn now to the consideration of buffering strategies.

5. BUFFERING STRATEGIES

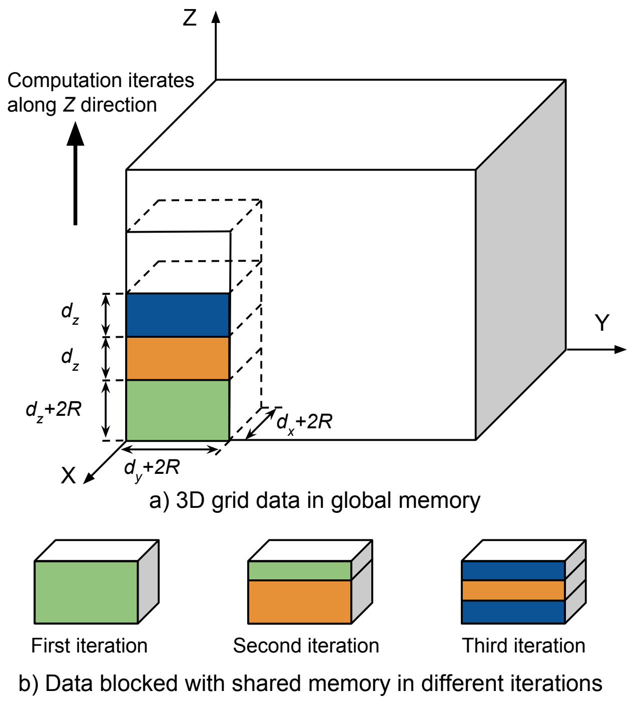
In this work, we will focus on the low level optimization for the GPU. In particular, we study three strategies of stencil code auto-generation, and the model-driven auto-tuning to the strategies. About the strategy that using read-only cache to buffer data, we leave it to our future work because of the page limitation. Read-only cache is hardware managed cache, to accuraly predict its performance we need to write micro-benchmarks to measure detailed cache parameters first, such as cache line size and set associtivity.

5.1 Baseline: no buffering

Because all the data for the computation is loaded from global memory, performance is tightly limited by global memory bandwidth. Although the performance of this method is relatively low, it uses no shared memory and the least registers. For multi-variable stencil computations when shared memory and registers limit occupancy, assigning a part of variables to be without buffering can increase occupancy, possibly increasing the overall performance.

5.2 Buffering with shared memory

Conventional shared memory buffering [1] maintains the data stored in shared memory to be in order. That is data fetched earlier is always stored at lower indices than data fetched later. This requires copying useful data from higher indices to lower indices before loading new data to the shared memory with every iteration. Such copying operation results in extra shared memory load and store operations. These copy operations are inefficient in that the ratio of overhead



Z
Computation iterates along Z direction
Y
X
d_z
d_z
d_z+2R
d_x+2R
d_y+2R
a) 3D grid data in global memory
First iteration
Second iteration
Third iteration
b) Data blocked with shared memory in different iterations

Figure 1: Illustration of buffering with shared memory for stencil computation that assigns a thread block with d_x, d_y, and d_z threads in the x, y and z dimension respectively. R denotes stencil radius.

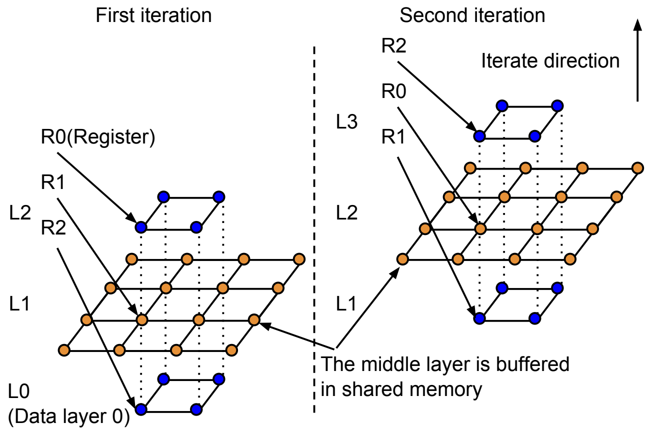


Figure 2: Illustration of buffering with shared memory and registers. In the second iteration, the register values of data layer 1 (i.e. L1) are from the shared memory of the first iteration. Similarly, the central shared memory values of data layer 2 (i.e. L2) are from the registers of the first iteration to minimize global memory accesses.

instructions to load and store instructions is high.

Figure 1 illustrates the buffering strategy we propose, which removes the requirement that memory be in order and thereby eliminates the data copying. Instead, we replace data that is no longer needed with newly loaded data.

5.3 Buffering with shared memory and registers

Figure 2 shows the strategy we call buffered with shared memory and registers. The middle layer is buffered in shared memory, while the others are in registers.

Buffering with shared memory would fetch all the data needed for an iteration into shared memory before computation. This requires a sufficiently large shared memory space and many shared memory load and store operations. For certain access patterns (sets of grid function offsets), such as "*plus*" and "*plus* $-$ *D*" access patterns, the data can be split between shared memory and registers, reducing both the amount of shared memory needed and shared memory instruction overhead.

We define the pattern of accesses to a grid function to be a plus pattern if no individual access has more than one nonzero offset. For example, the offset (x, y, z) can only be of the format $(x, 0, 0)$, $(0, y, 0)$, or $(0, 0, z)$, where x, y, and z, can be any integers. We define a pattern to be a plus-D pattern, where D is a coordinate axis, if there is no individual access in which the offsets in both the D dimension and some other dimension are non-zero. For example, the offset (x, y, z) of the plus-Z pattern can only be in the format of $(x, y, 0)$ or $(0, 0, z)$.

A mixed register/shared memory strategy can be applied to plus-D pattern grid functions on block configurations with one thread in the D dimension (with D being either Y or Z), and in which each iteration proceeds along the D dimension. Shared memory will only be used for the layer of data which is orthogonal to the D dimension. Other values will be placed in local memory, with the expectation that the compiler will use registers for these values. This mixed register/shared memory storage reduces shared memory usage and assignment to a register avoids the need for a separate shared memory load instruction.

6. PERFORMANCE MODEL

We will introduce the model for a single thread block specific to the applied code auto-generation strategy, then the model for the total execution time.

6.1 Modeling of a single thread block

Figure 3 shows the time for executing a thread block. We assume all streaming multiprocessors (SMs) are running with the same block configuration C_{Block}:

$$C_{\text{Block}} = (d_x, d_y, d_z, N_{\text{iter}}, \text{dir}_{\text{iter}}) \qquad (1)$$

We define C_{Block} as above, in which d_x, d_y, and d_z threads are assigned to a thread block in the x, y and z dimension respectively. N_{iter} denotes the number of stencil points each thread computes. The symbol dir_{iter} denotes the direction along which the computation iterates (in this work we always iterate along the z direction). Note that below our time units are in clock cycles, and so times must be reported as integers.

The execution time of a thread block can be defined as below:

$$T(C_{\text{Block}}) = T_{\text{init}} + T_{\text{iter\_1st}} + (N_{\text{iter}} - 1)T_{\text{iter\_oth}} + T_{\text{diff}} \quad (2)$$

This same equation will be applied for all three of the strategies we describe in this paper, the unbuffered, the buffered with shared memory, and the buffered with shared memory and registers strategy.

- Without buffering: $T_{\text{iter\_1st}} = T_{\text{iter\_oth}}$

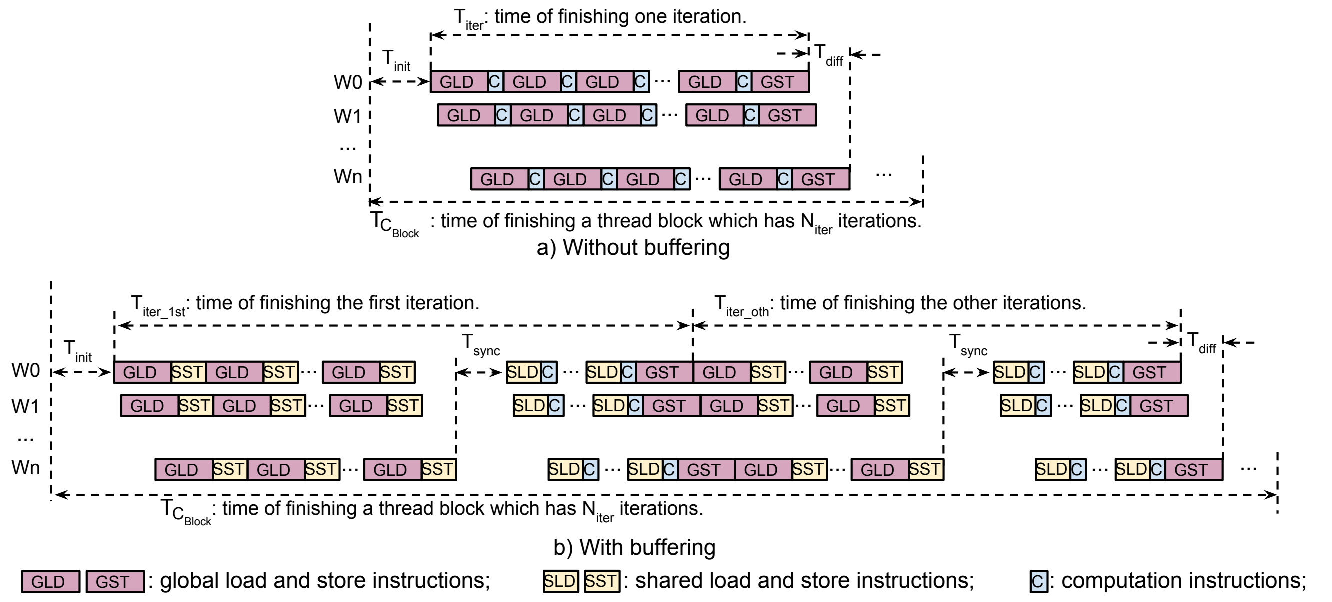

T iter: time of finishing one iteration.
T init
T diff
W0
W1
...
Wn
GLD
C
GST
T C Block : time of finishing a thread block which has N iter iterations.
a) Without buffering
T iter_1st: time of finishing the first iteration.
T iter_oth: time of finishing the other iterations.
T sync
SST
SLD
b) With buffering
GLD GST: global load and store instructions;
SLD SST: shared load and store instructions;
C: computation instructions;

Figure 3: Execution of a single thread block at the instruction level, both without (a) and with (b) buffering. Warps are numbered W0 to Wn, and time is not meant to scale between the subplots.

- With buffering: $T_{\text{iter\_1st}} > T_{\text{iter\_oth}}$

T_{init} denotes the time spent on initialization, including address calculation etc. T_{iter} denotes the time it takes to finish one iteration. With buffering, the first iteration ($T_{\text{iter\_1st}}$) takes longer time than the others ($T_{\text{iter\_oth}}$) as it needs to load more data for buffering. We will discuss the calculation of iteration time in Section 6.1.1. In Section 6.1.5, we will discuss T_{diff}, which denotes the time difference between the first and last warp of an iteration.

$$T_{\text{iter}} = T_{\text{gld}} + T_{\text{sst}} + T_{\text{sync}} + T_{\text{sld}} + T_{\text{comp}} + T_{\text{gst}} \qquad (3)$$

The iteration time T_{iter} is modeled as the sum of global load time T_{gld}, shared store time T_{sst}, thread synchronization time T_{sync}, shared load time T_{sld}, total computation time T_{comp}, and global store time T_{gst}. T_{sst}, T_{sync}, and T_{sld} are zero for the unbuffered case. For the buffered case, we will discuss the computation of T_{sst}, T_{sync}, and T_{sld} in Section 6.1.2, 6.1.3, and 6.1.4 separately.

We assume the latency of floating point instructions can be hidden. Therefore T_{comp} is determined by the throughput of floating point units. Each SM can execute 64 double precision floating point operations per cycle (i.e. $\theta_{\text{fp}} = 64$). That means that for all the warps(N_{warp}) to complete N_{flop} floating point operations through the SM will take T_{comp} cycles. N_{flop} denotes the number of data elements needed to compute a stencil point (see Table 2). $N_{\text{thread/warp}}$ denotes the number of threads in a warp, which is 32[18].

$$T_{\text{comp}} = N_{\text{flop}} \left\lceil \frac{N_{\text{warp}}}{\theta_{\text{fp}}/N_{\text{thread/warp}}} \right\rceil \qquad (4)$$

6.1.1 *Computation of* T_{gld} *and* T_{gst}

The most complicated part of this calculation is T_{gld} which we give in general form in Eq. 5. The symbol p denotes the number of terms needed for the corresponding computation. For the unbuffered strategy, there is only one term to consider. For shared memory, however, we read in central or boundary regions independently, and each of these loads corresponds to a term.

$$T_{\text{gld}} = \left\lceil \frac{\sum_{i=1}^{p} N_{\text{mem}}^{(i)}}{\lambda} \right\rceil \times T_{\text{gmem\_latency}} + \sum_{i=1}^{p} \left\lceil \frac{N_{\text{thd}}^{(i)} N_{\text{mem}}^{(i)}}{\theta_{\text{gmem/sm}} \beta_{\text{gm}}} \right\rceil \qquad (5)$$

Table 1 lists the model parameters for loading data from global memory. $N_{\text{thd}}^{(i)}$ denotes the number of threads assigned for such loading. $N_{\text{mem}}^{(i)}$ denotes the number of data elements each thread will load. For the buffered case, as the loaded data is stored to shared memory, it will also be used to compute T_{sst} in Section 6.1.2. $\theta_{\text{gmem/sm}}$ denotes the global memory bandwidth per SM. $N_{\text{thd\_x}}^{(i)}$ denotes the number of threads along the X direction for loading data from global memory.

We define β_{gm} to be the global memory request utilization, which is modeled as the ratio of the requested data size over the actual transferred data size.

The parameters used, by model, for this equation are supplied in Table 1. In Table 1, we use the following definitions:

$$d^3 = d_x d_y d_z \qquad (6)$$
$$r_x = d_x + 2R \qquad (7)$$
$$r_y = d_y + 2R \qquad (8)$$
$$r_z = d_z + 2R \qquad (9)$$
$$d_{\text{cut}} = \left\lfloor \frac{d^3}{r_x} \right\rfloor r_x \qquad (10)$$
$$d_{cmin} = \min(d_{\text{cut}}, r_x r_y) \qquad (11)$$

For the unbuffered case, $N_{\text{thd}} = d_x \times d_y \times d_z$ and $N_{\text{mem}} = N_{\text{elem}}$ (the value of N_{elem} depends on the stencil. See Table 2).

The term T_{gld} describes the time it takes to load from

strategy	iter	p	i	$N_{\text{thd}}^{(i)}$	$N_{\text{mem}}^{(i)}$
unbuffered	all	1	1	d^3	N_{elem}
buffered with shared memory	1st	2	1	d_{cmin}	$\left\lfloor \frac{r_x r_y}{N_{\text{thd}}^{(1)}} \right\rfloor r_z$
			2	$r_x r_y \mod N_{\text{thd}}^{(1)}$	r_z
	other	2	1	d_{cmin}	$\left\lfloor \frac{r_x r_y}{N_{\text{thd}}^{(1)}} \right\rfloor d_z$
			2	$r_x r_y \mod N_{\text{thd}}^{(1)}$	d_z
buffered with shared memory and registers	1st	3	1	d_{cmin}	$\left\lfloor \frac{r_x r_y}{N_{\text{thd}}^{(1)}} \right\rfloor d_z$
			2	$r_x r_y \mod N_{\text{thd}}^{(1)}$	d_z
			3	d^3	$2R$
	other	5	1	d^3	1
			2	$\min(d_{\text{cut}}, r_x R)$	$2 \left\lfloor \frac{r_x R}{N_{\text{thd}}^{(2)}} \right\rfloor$
			3	$r_x R \mod N_{\text{thd}}^{(2)}$	$N_{\text{mem}}^{(2)}$
			4	$\min(d_{\text{cut}}, d_y R)$	$2 \left\lfloor \frac{d_y R}{N_{\text{thd}}^{(4)}} \right\rfloor$
			5	$d_y R \mod N_{\text{thd}}^{(4)}$	$N_{\text{mem}}^{(4)}$

Table 1: Model parameters for computing global and shared memory latencies. Steps with $N_{\text{thd}}^{(i)} = 0$ do not need to execute.

global memory. N_{elem} denotes the number of data elements needed to compute a stencil point. We define the maximum number of elements that can be loaded per thread in parallel as λ. The value of λ is hard to model as it is determined by the number of hardware registers and how the compiler allocates them. For the K20xm, we assume λ is equal to 4.

The calculation of T_{gst} is similar to that of T_{gld}. The only difference is that N_{elem} is fixed to be 1 as we store results back once every iteration.

6.1.2 Computation of T_{sst}

$$T_{\text{sst}} = \frac{\sum_{i=1}^{p} N_{\text{elem}}^{(i)} \left\lceil \frac{N_{\text{thd}}^{(i)}}{N_{\text{thd/warp}}} \right\rceil \beta_{\text{sm}}}{\theta_{\text{sm}}} \tag{12}$$

The quantity β_{sm} is the maximum number of times a store operation will fall into the same shared memory bank within the same warp during a kernel computation, and so can be considered a measure of contention. The minimum value of β_{sm} is 1, which is also the ideal value. We denote θ_{sm} as the shared memory access throughput. For Kepler GPUs, $\theta_{\text{sm}} = 1$ as a warp can have a shared memory access in every cycle[18].

6.1.3 Computation of T_{sync}

All threads of a thread block have to synchronize before sharing data to avoid race conditions. We estimate synchronization time by following previous work [17] as some latency plus a time proportional to the number of warps:

$$T_{\text{sync}} = T_{\text{gmem\_lat}} + \alpha N_{\text{warp}} \, . \tag{13}$$

We assume α to be 10 cycles in this work for simplicity. The dependency of α on a particular GPU is left to future work.

6.1.4 Computation of T_{sld}

$$T_{\text{sld}} = N_{\text{sld}} \times \beta_{\text{sm\_sld}} \times T_{\text{sm\_lat}} \tag{14}$$

For the strategy buffered with shared memory, the number of shared loads T_{sld} it takes is equal to the number of elements (N_{elem} in Table 2) each thread needs to compute a stencil point. For the strategy buffered with shared memory and registers, the data along Z direction is buffered in registers. Take the benchmark for the SPLUS stencil in Table 2 for example, N_{sld}, the number of shared loads needed for the stencil points surrounding the center is $N_{\text{sld}} = 4 \times R$ where R is the stencil radius.

6.1.5 Computation of T_{diff}

The time difference has four potential sources: warp issuing ($T_{\text{diff\_issue}}$), floating point computation ($T_{\text{diff\_comp}}$), shared memory access ($T_{\text{diff\_shmem}}$), and global memory access. The actual time difference is no less than their maximum depending on how well they overlap. We do not consider the time difference from global memory since an accurate modeling requires a detailed memory simulator, and it is not important in this work. We also assume the times will perfectly overlap.

$$T_{\text{diff}} = \max(T_{\text{diff\_issue}}, T_{\text{diff\_comp}}, T_{\text{diff\_shmem}}) \tag{15}$$

NVIDIA Kepler GPUs have four warp schedulers (i.e. $N_{\text{warp\_scheduler}} = 4$). Assuming a warp scheduler can issue one warp per cycle, the time it takes to issue N_{warp} warps is:

$$T_{\text{diff\_issue}} = \left\lceil \frac{N_{\text{warp}}}{N_{\text{warp\_scheduler}}} \right\rceil \tag{16}$$

Each SM can execute 64 double precision floating point operations per cycle (i.e. $\theta_{\text{fp}} = 64$). That means that for all the threads to feed one floating point operation through the SM will take $T_{\text{diff\_comp}}$ cycles.

$$T_{\text{diff\_comp}} = \left\lceil \frac{N_{\text{warp}}}{\theta_{\text{fp}} / N_{\text{thread/warp}}} \right\rceil \tag{17}$$

Only one warp is allowed to access shared memory per cycle. If there is no shared memory access, $T_{\text{diff\_shared\_mem}} = 0$. Otherwise,

$$T_{\text{diff\_shmem}} = \beta_{\text{sm}} \times N_{\text{warp}} \tag{18}$$

6.2 Modeling of total execution time

We consider partial executions while modeling total execution time. A partial execution happens when some threads of a thread block are idle. For a given block configuration C_{Block} (see Eq. 1), the number of stencil points it computes are d_x, d_y, and $d_z \times N_{\text{iter}}$, along the x, y, and z direction. There are eight types of C_{Block} corresponding to whether ($D_x \mod d_x$), ($D_y \mod d_y$), and ($D_z \mod d_z$) are zero or not. For a user provided block configuration C_{Block}, we consider the non-partial (C_{Block0}), plus all 7 possible partial executions (C_{Blocki}), for $i = 1, \ldots, 7$ during execution.

Let $N_{\text{C}_{\text{Block}_\text{i}}}$ denote the number of type i block configurations. Let $T_{\text{C}_{\text{Block}_\text{i}}}$ denote the execution time of a block with type i block configuration ($i = 0, 1, \ldots, 7$). Note in our code auto-generation framework, each SM only has one thread block (i.e. $N_{\text{block/sm}}$=1) to maximize the amount of sharing. Let N_{SM} denotes the number of SMs the GPU has, then the total execution time can be modeled using the

Table 2: Benchmark characteristics

	SPLUS	DPLUS	THB	CUBE
N_{elem}	$1+6R$	$1+6R$	$(1+2R)^2+2R$	$(1+2R)^3$
N_{flop}	$1+6R$	$2\times(1+6R)$	$(1+2R)^2+2R$	$(1+2R)^3$

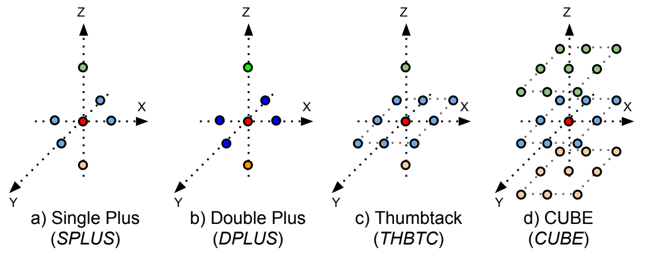


Figure 4: Evaluated benchmarks. (For DP, each point is used twice for computation.)

equation below:

$$T_{\text{total}} = \frac{\sum_{i=0}^{7} N_{\text{C}_{\text{Block}_\text{i}}} \times T_{\text{C}_{\text{Block}_\text{i}}}}{N_{\text{SM}} \times N_{\text{block/sm}}} \qquad (19)$$

6.3 GPU-Dependent Parameters

The model contains explicit and implicit GPU-dependent parameters. The value of explict parameters, such as the number of double precision floating point operations per cycle (i.e. $\theta_{\text{fp}} = 64$), are explicitly defined by NVIDIA[18]. We assume the value of the implicit parameters by analyzing assembly code and performance profiling results. We will use micro-benchmarks to better decide these values in our future work.

7. EXPERIMENTAL METHODOLOGY

We evaluated our auto-tuning system on nodes of the LSU machine Shelob, equipped with two Intel Xeon E5-2670 processors, one NVIDIA Kepler K20Xm GPU, and 64 GiB RAM.

7.1 Evaluated benchmarks

In this work we evaluate 3D stencil computations which have 256 stencil points along each dimension ($D_x = D_y = D_z = 256$), using four types of stencil benchmarks extracted from real scientific computation, see Figure 4. Table 2 lists the characteristics of the evaluated benchmarks. R denotes the radius of the stencil. N_{elem} and N_{flop} denote the number of data elements and the number of double-precision floating point operations (counting fused multiply-adds as one operation) it takes to compute a single stencil point.

7.2 Evaluation method

We ran experiments to determine how the predictions of our auto-tuning system compared to actual executions for choosing the best configuration.

For problem size ($D_x = D_y = D_z = 256$), there are 3,659,653 possible block configurations. We randomly select 200 *runnable* block configurations for evaluation. A block is called runnable if there is sufficient shared memory for the configuration. For the shared memory and registers strategy, the number of threads along iteration direction has to be one. For each of these two hundred cases, we compare the measured performance with the model predicted performance.

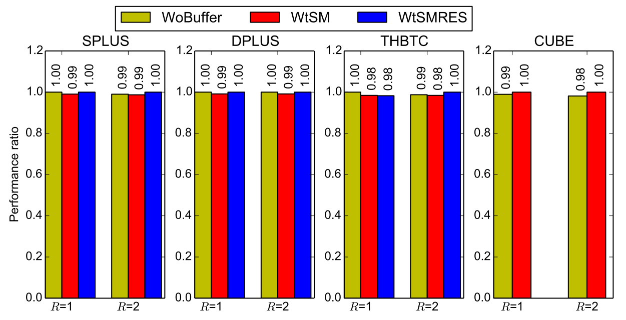


Figure 5: Performance difference between the predicted best and the measured best block configurations.

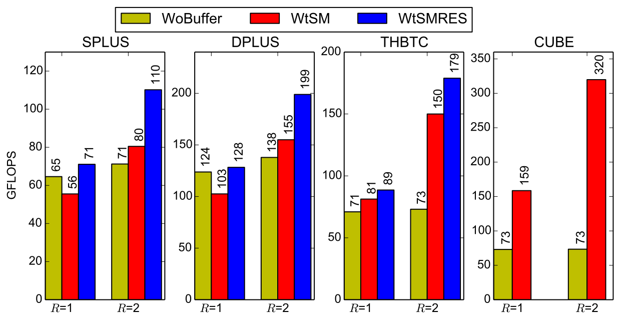


Figure 7: Best performance of auto-generated stencil code achieved by pre-running 200 random block configurations and selecting the best one.

8. RESULTS

For all stencil shapes considered, our predicted and measured performance are in high agreement, especially for the predicted best block configurations for which the predicted and measured performance differs by only a few percent. Note we normalize the performance for each strategy in Figure 6 using a factor of 1.5 for the WoBuffer, 3.1 for WtSM, and 2.5 for WtSMRES. These were empirically determined and were necessary because we cannot model every coding detail, and because uncertainties exist in parameters such as the global memory latency. Note that these normalization factors are relevant for choosing between models, but are not relevant for picking the best configuration within a model.

8.1 Performance of code auto-tuning

Figure 5 illustrates the performance ratio of the predicted best block configuration over the measured best block configuration. As we can see, for 11 out of 22 auto-tuning tests(like benchmark SPLUS, strategy WoBuffer, R=1), the predicted best block configuration is the same as the measured best block configuration. For the other 11 auto-tuning tests, the predicted best block configuration achieves more than 98.13% of the actual best performance. For benchmark CUBE, strategy WtSMRES is not shown as it only applies

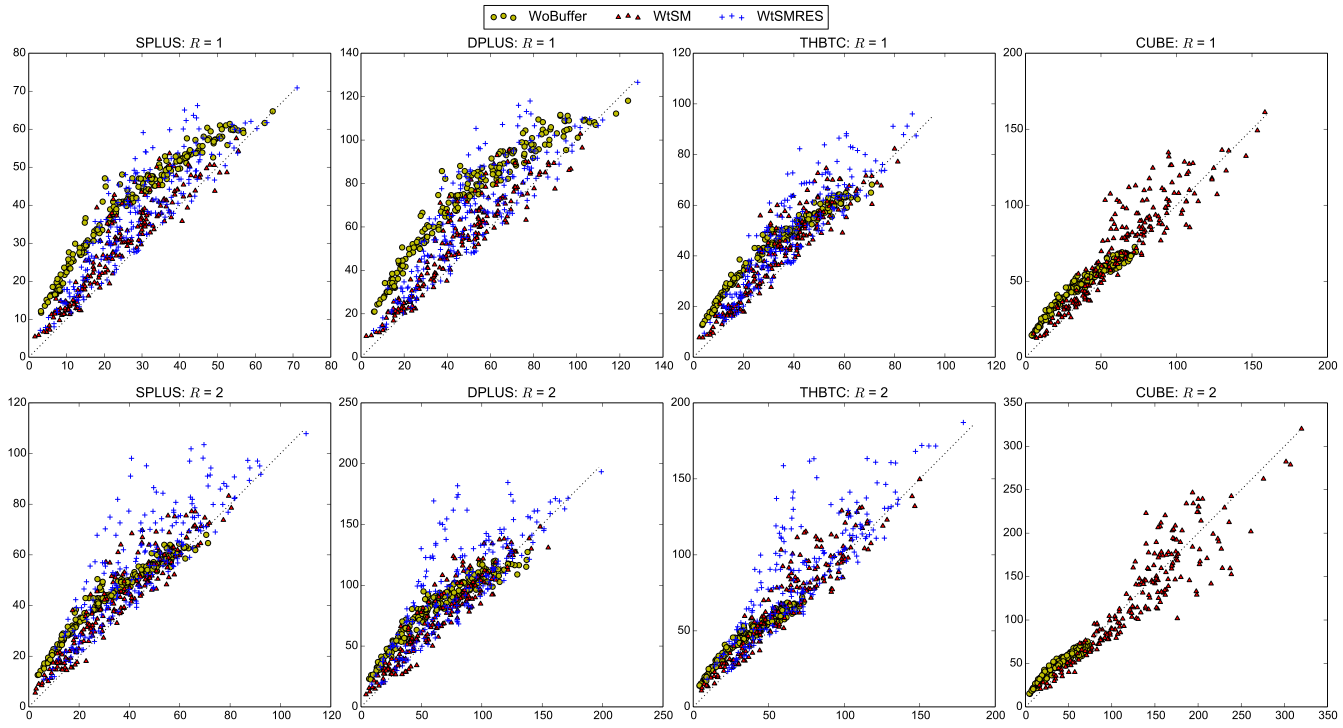


Figure 6: Evaluating performance model on three code auto-generation strategies (WoBuffer, WtSM, and WtSMRES) with four stencil benchmarks (SPLUS, DPLUS, THBTC, and CUBE), two R values, and 200 randomly selected block configurations. The horizontal and vertical axes are the measured and predicted performance expressed in GFLOPS.

to "$plus$" and "$plus - D$" access patterns, while the access pattern of benchmark CUBE is neither "$plus$" nor "$plus - D$" as discussed in Section 5.3.

Figure 6 shows detailed results of the predicted (vertical axis) and measured (horizontal axis) performance for all evaluated block configurations. There are 200 block configurations for all strategies of each sub figure, in which a block configuration is shown as a marker. The performance model predicts the block configurations that have larger value along vertical axis would achieve better performance. In other words, the model predicts the block configuration that has the largest value along vertical axis to be the best block configuration. For each block configuration, the value along horizontal direction denotes the measured performance. We can see, our performance model is accurate enough for auto-tuning, as the markers generally locates on the dotted straight line in the figure. This is especially true for the predicted best block configurations.

8.2 Performance of code auto-generation

Figure 7 illustrates the best performance of the auto-generated stencil code. For strategy WoBuffer, the best performance of all benchmarks except DPLUS, is around 65-73 GFLOPS as it is tightly bounded by global memory bandwidth. As global memory bandwidth is 250 GB/s and there is one Fused Multiply-Add operation (2 flops) on each 8-byte data element, the expected peak performance for strategy WoBuffer is:

$$\frac{2 \times 250 \times 1024^3}{8 \times 10^9} = 67.11\,\mathrm{GFLOPS}$$

The performance of benchmark DPLUS nearly double what is achieved by SPLUS as each point is used twice for computing a single stencil point. For the buffered strategies WtSM and WtSMRES, the performance improvement over unbuffered strategy WoBuffer depends on how much global memory accesses can be reduced and buffering overhead such as extra shared memory load and store instructions. Generally, higher N_{flop} value in Table 2 means more global memory accesses can be reduced for the buffered strategies. Strategy WtSMRES achieves better performance than strategy WtSM as it reduces shared memory load and store operations by buffering some data in registers.

9. CONCLUSION

Execution driven auto-tuning puts tight limits on the size of the parameters space that can be explored. In this work we demonstrate model driven auto-tuning for GPUs which avoids the delay of execution driven model execution. This opens the possibility for exploring a richer configuration space for auto-tuning, such as assigning an access function (e.g. global or shared memory) to each computational variable, even when each variable has a different type of stencil.

These techniques will be used in the Chemora code generation and optimization framework for differential equations, in which we plan for auto-tuning to encompass the higher level parts of the system up to discretization methods.

In this paper we have demonstrated a first step toward this goal by showing how to model and tune a configuration for a single variable and multiple stencil shapes and radii.

10. ACKNOWLEDGMENTS

We thank Peter Diener, Roland Haas, Ian Hinder, and Jian Tao as the main authors of Chemora. Cactus and Chemora are currently supported by NSF awards 0905046 *PetaCactus: Unraveling the supernova — Gamma-ray burst mystery*; 0941653 *Enabling Science at the Petascale: From Binary Systems and Stellar Core Collapse To Gamma-Ray Bursts*; 1212401, 1212426, 1212433, 1212460 *The Einstein Toolkit — An open source general relativistic multi-physics infrastructure for relativistic astrophysics*; 1265449, 1265434, 1265451 *Using PDE descriptions to generate code precisely tailored to energy-constrained systems including large GPU accelerated clusters*. We would also like to thank NVIDIA Corporation for providing some of the GPUs used for this study.

Our research was made possible by access to HPC resources at Louisiana State University, in particular the Shelob cluster using the allocations hpc_numrel and hpc_hyrel.

References

[1] Paulius Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[2] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE Computer Society, 2010.

[3] Naoya Maruyama and Takayuki Aoki. Optimizing stencil computations for NVIDIA Kepler GPUs. *HiStencils 2014*, page 89, 2014.

[4] J Porter-Sobieraj, S Cygert, D Kikoła, J Sikorski, and M Słodkowski. Optimizing the computation of a parallel 3D finite difference algorithm for graphics processing units. *Concurrency and Computation: Practice and Experience*, 2014.

[5] David Michéa and Dimitri Komatitsch. Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards. *Geophysical Journal International*, 182(1):389–402, 2010.

[6] Matthias Christen, Olaf Schenk, Esra Neufeld, Peter Messmer, and Helmar Burkhart. Parallel data-locality aware stencil computations on modern micro-architectures. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009.

[7] Chemora project website. `http://chemoracode.org/index.php/Main_Page`.

[8] Marek Blazewicz, Ian Hinder, David M Koppelman, Steven R Brandt, Milosz Ciznicki, Michal Kierzynka, Frank Löffler, Erik Schnetter, and Jian Tao. From physics model to results: An optimizing framework for cross-architecture code generation. *Scientific Programming*, 21(1):1–16, 2013.

[9] Marek Blazewicz, Steven R Brandt, Peter Diener, David M Koppelman, Krzysztof Kurowski, Frank Löffler, Erik Schnetter, and Jian Tao. A massive data parallel computational framework for petascale/exascale hybrid computer systems. *arXiv preprint arXiv:1201.2118*, 2012.

[10] Erik Schnetter. Performance and optimization abstractions for large scale heterogeneous systems in the cactus/chemora framework. In *Extreme Scaling Workshop (XSW), 2013*, pages 33–42. IEEE, 2013.

[11] Jian Tao, Marek Blazewicz, and Steven R Brandt. Using gpu's to accelerate stencil-based computation kernels for the development of large scale scientific applications on heterogeneous systems. In *ACM SIGPLAN Notices*, volume 47, pages 287–288. ACM, 2012.

[12] Justin Holewinski, Louis-Noël Pouchet, and P Sadayappan. High-performance code generation for stencil computations on GPU architectures. In *Proceedings of the 26th ACM international conference on Supercomputing*, pages 311–320. ACM, 2012.

[13] Thibaut Lutz, Christian Fensch, and Murray Cole. Partans: An autotuning framework for stencil computation on multi-gpu systems. *ACM Transactions on Architecture and Code Optimization (TACO)*, 9(4):59, 2013.

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

[15] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[16] Jiayuan Meng and Kevin Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*, pages 256–265. ACM, 2009.

[17] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 152–163, New York, NY, USA, 2009. ACM.

[18] CUDA programming guide 6.5. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz3HCufb6HN`.