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
[2]School of Electrical Engineering and Computer Science, Louisiana State University
[3]Department of Computer Science, Louisiana State University

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

January 20, 2015

Overview

1. Motivation
2. Performance Model
3. Experimental Methodology
4. Results Analysis
5. Conclusion

Motivation

- Stencil code auto-generation is widely used in scientific computing
- Execution-driven auto-tuning is time-consuming
  - 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
- We propose an analytic performance model for stencil codes on GPUs
  - Deliver close-to-optimal performance
  - Not require extensive tuning at compile or run time
  - Reduce auto-tuning time by more than 10,000 times

This work

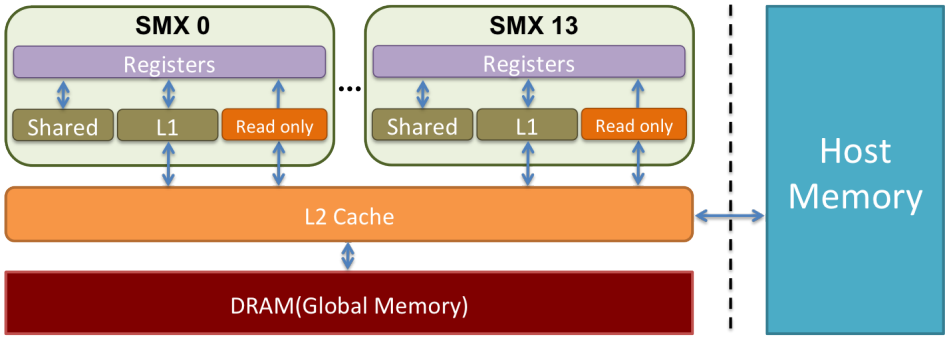
- Implement three code auto-generation strategies
- Build an analytic performance model to help select the best configuration without the needs for test runs
- Consider instruction throughput, off-chip memory bandwidth, exposed latencies, and contentions of variable types
- Evaluate the model with four types of stencil benchmarks extracted from real scientific computation

As a start:

- Focus on one-variable stencil computation
- Will study read-only cache strategy in future work as:
  - Hardware-managed cache
  - Accurate performance modeling requires to measure detailed cache parameters (e.g. set associtivity) first
  - We are working on related micro-benchmarks for the measurements

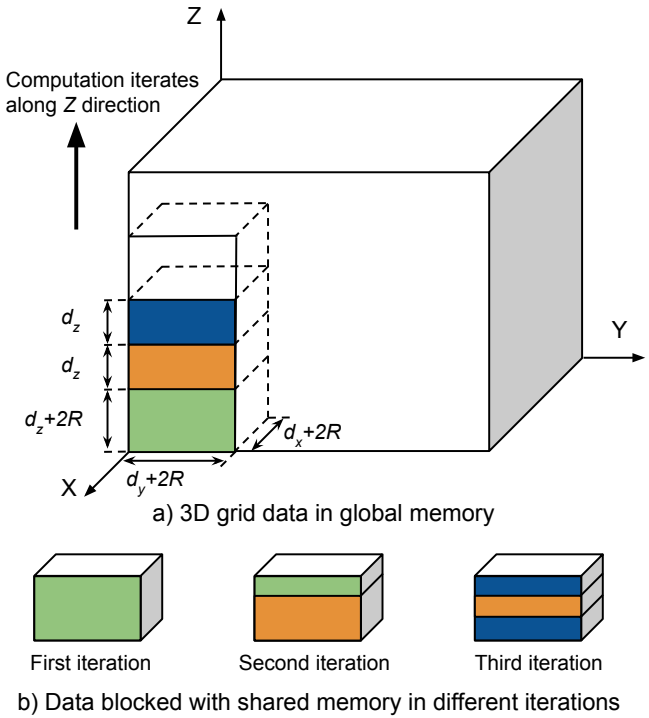
Stencil code auto-generation

Memory hierachy of NVIDIA Kepler GPUs:



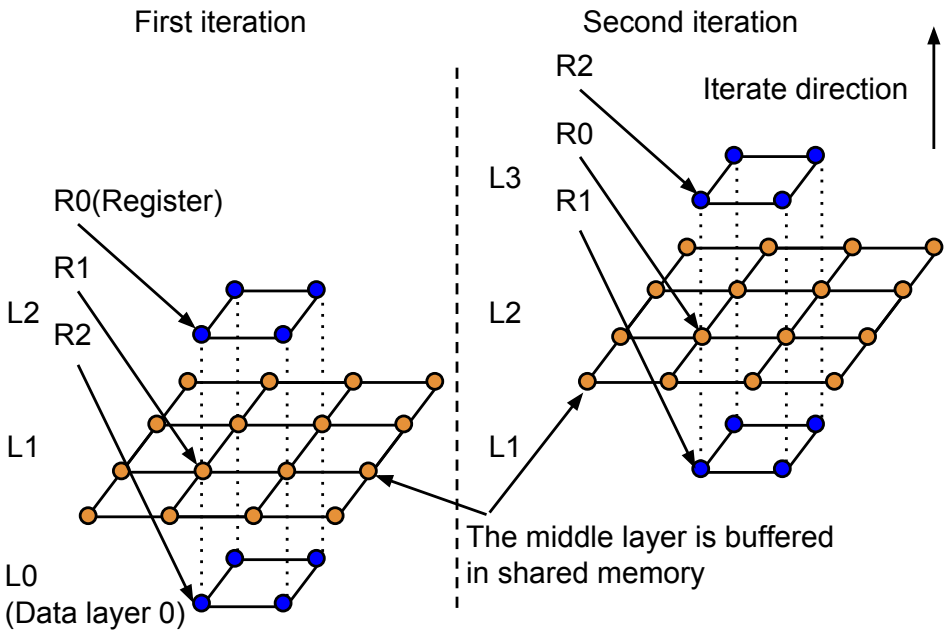
- Implement 3 code-generation strategies: a) without buffering; b) with shared memory; c) with shared memory and registers
- For multi-variable stencil computations, different variables can apply different strategies
- Auto-tuning is to decide which strategy and block onfiguration $(d_x, d_y, d_z, N_{\text{iter}}, \text{dir}_{\text{iter}})$ would achieve the best performance
  - 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
  - dir_{iter} denotes the direction along which the computation iterates

Strategy: buffering with shared memory



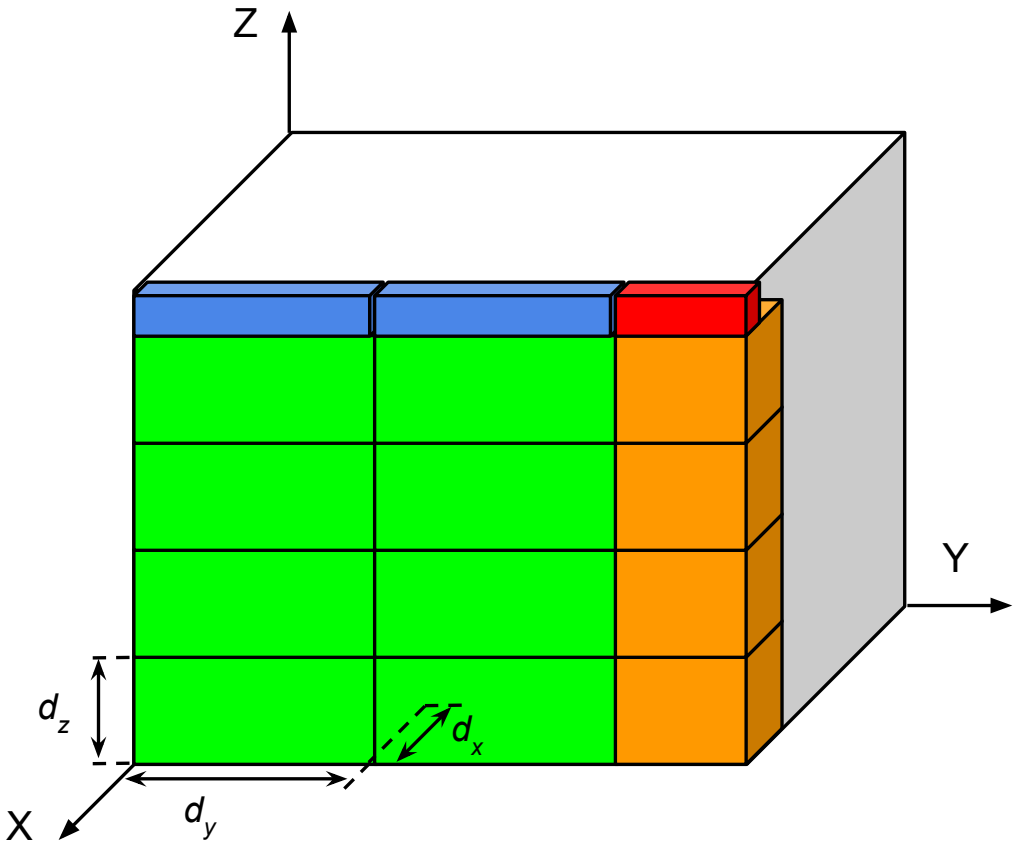
- A thread block with d_x, d_y, and d_z threads along X, Y, Z direction. R denotes stencil radius
- Shared memory size: $(d_x + 2R) \times (d_y + 2R) \times (d_z + 2R)$
- Replace data that is no longer needed with newly loaded data to reduce shared memory overhead

Strategy: buffering with shared memory and registers



- Registers are the fastest on-chip memory
- The middle layer is buffered in shared memory, while the others in registers
- This reduces the needed shared memory and its instruction overhead

Partial execution



Non-partial execution: all threads are used

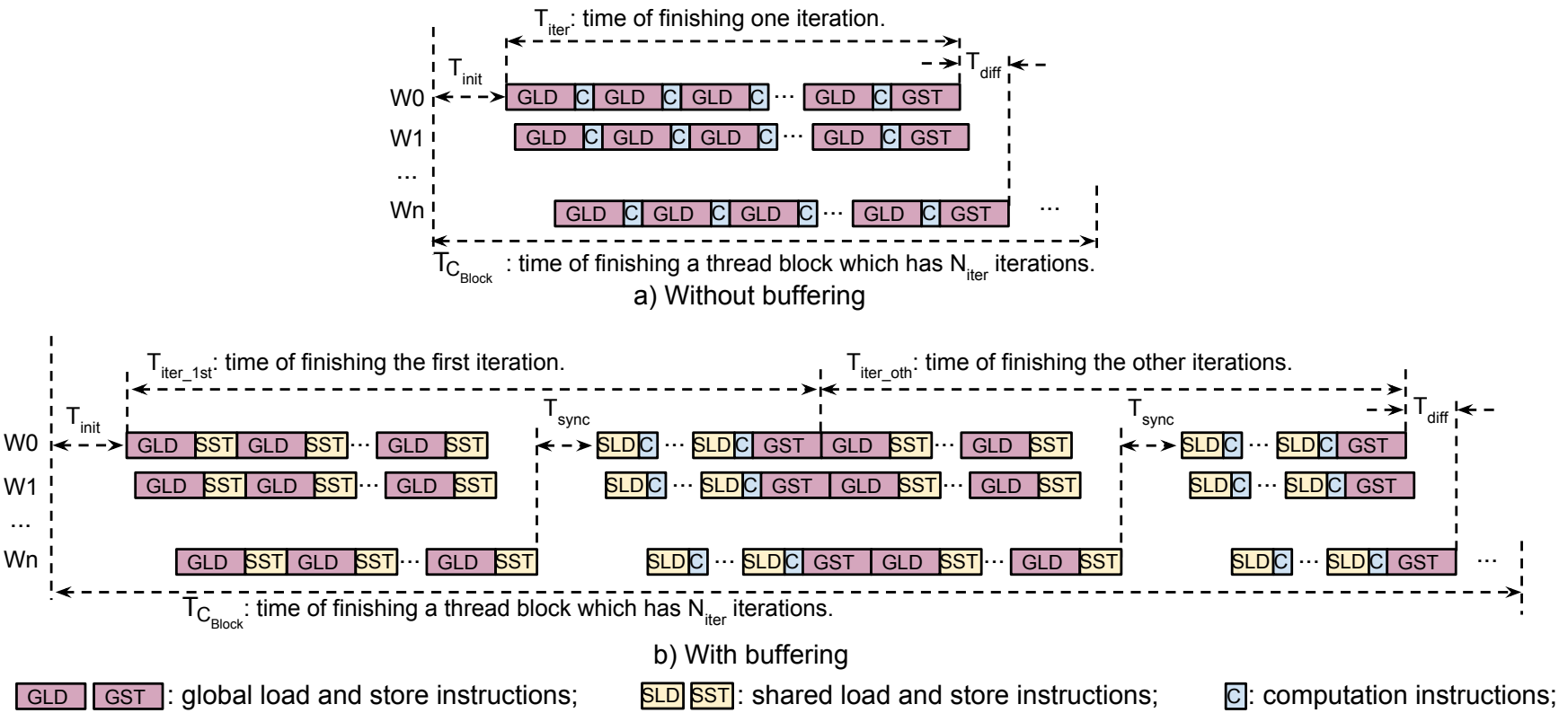
Partial execution: some threads are idle

Modeling of total execution time

$$T_{\text{total}} = \frac{\sum_{i=0}^{7} N_{\text{C}_{\text{Block}_\text{i}}} \times T_{\text{C}_{\text{Block}_\text{i}}}}{N_{\text{SM}} \times N_{\text{block/sm}}} \tag{1}$$

- For a user provided block configuration C_{Block}, we consider the non-partial (C_{Block0}), plus all 7 possible partial executions ($C_{\text{Block}_\text{i}}$), for $i = 1, \ldots, 7$ during execution.A partial execution happens when some threads of a thread block are idle.
- $N_{\text{C}_{\text{Block}_\text{i}}}$: the number of type i block configurations.
- $T_{\text{C}_{\text{Block}_\text{i}}}$: execution time of a block with type i block configuration.
- $N_{\text{block/sm}}$: the number of threads blocks per SM
- N_{SM}: the number of SMs the GPU has

Modeling of a single thread block



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

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

Modeling of the iteration time T_{iter}

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

- T_{gld}: global load time
- T_{sst}: shared store time
- T_{sync}: thread synchronization time
- T_{sld}: shared load time
- T_{comp}: total computation time
- T_{gst}: global store time

Modeling of global load time T_{gld}

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

- p: the number of terms needed for the corresponding computation
- $N_{\text{mem}}^{(i)}$: the number of data elements each thread will load
- $N_{\text{thd}}^{(i)}$: the number of threads assigned for such loading
- λ: the maximum number of elements that can be loaded per thread in parallel
- $\theta_{\text{gmem/sm}}$: the global memory throughput per SM
- β_{gm}: global memory request utilization, modeled as the ratio of the requested data size over the actual transferred data size
- Modeling of global store time is similar

Modeling of others

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

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

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

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

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

$$d^3 = d_x d_y d_z \quad (6)$$

$$r_x = d_x + 2R \quad (7)$$

$$r_y = d_y + 2R \quad (8)$$

$$r_z = d_z + 2R \quad (9)$$

$$d_{\text{cut}} = \left\lfloor \frac{d^3}{r_x} \right\rfloor r_x \quad (10)$$

$$d_{cmin} = \min(d_{\text{cut}}, r_x r_y) \quad (11)$$

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

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

$$T_{\text{sld}} = N_{\text{sld}} \times \beta_{\text{sm\_sld}} \times T_{\text{sm\_lat}} \quad (14)$$

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

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

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

$$T_{\text{diff\_shmem}} = \beta_{\text{sm}} \times N_{\text{warp}} \quad (18)$$

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

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	$d_y R \mod N_{\text{thd}}^{(4)}$	$N_{\text{mem}}^{(4)}$

In addition, we modeled:

- Coalesced access to global memory
- Bank conflict in shared memory

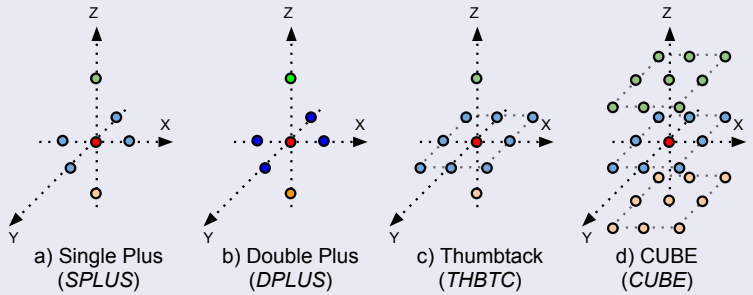
See paper for details.

GPU-Dependent Paramters

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. 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.

Experimental Methodology

Benchmarks



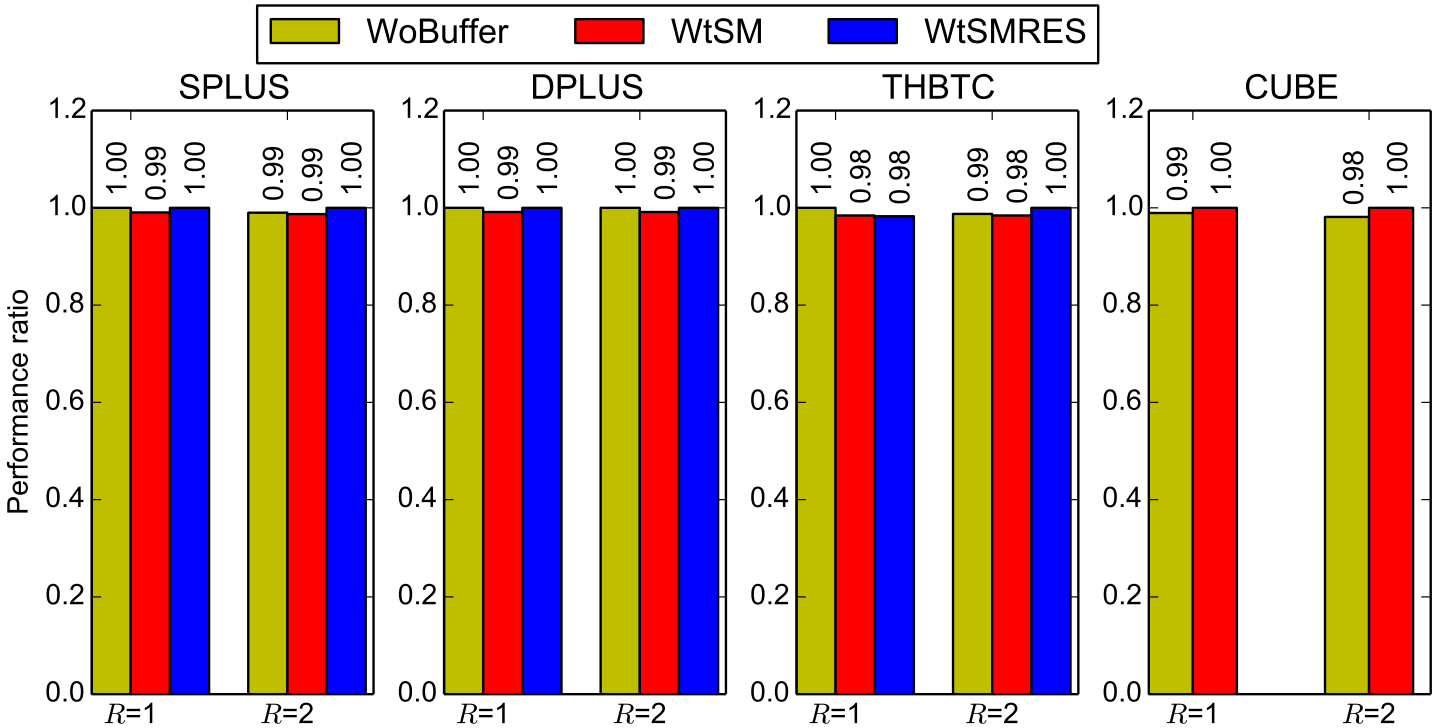
	SPLUS	DPLUS	THBTC	CUBE
N_{elem}	$1+6R$	$1+6R$	$(1+2R)^2+2R$	$(1+2R)^3$
N_{flop}	$1+6R$	$2\times(1+6R)$	$(1+2R)^2+2R$	$(1+2R)^3$

- R: the radius of the stencil.
- N_{elem} and N_{flop}: # of data elements and # of floating point operations it takes to compute a single stencil point.

Evaluation method

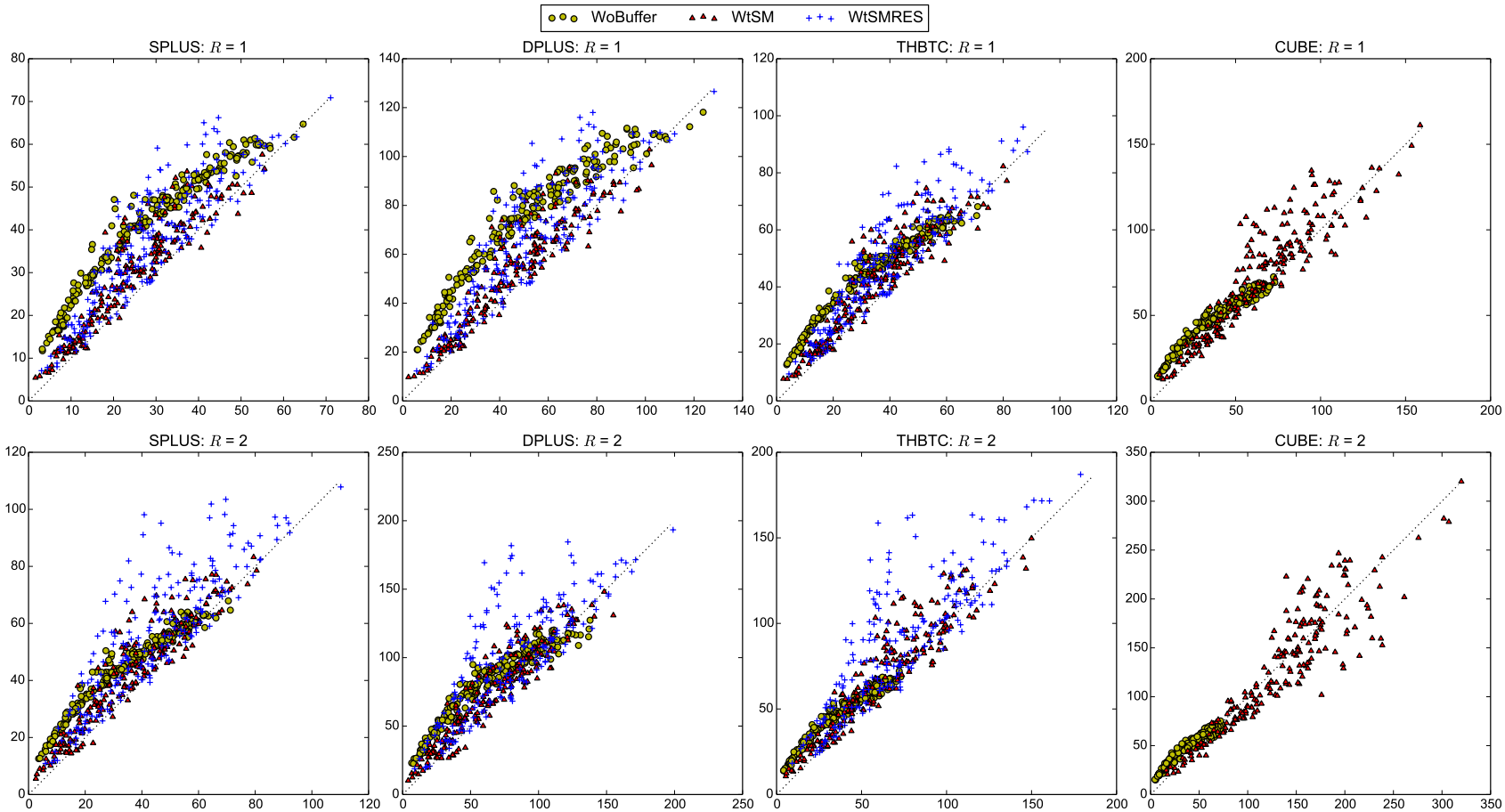
- Randomly select 200 runnable block configurations
- Run experiments to observe the performance difference between the prediction and the actual execution.

Performance prediction



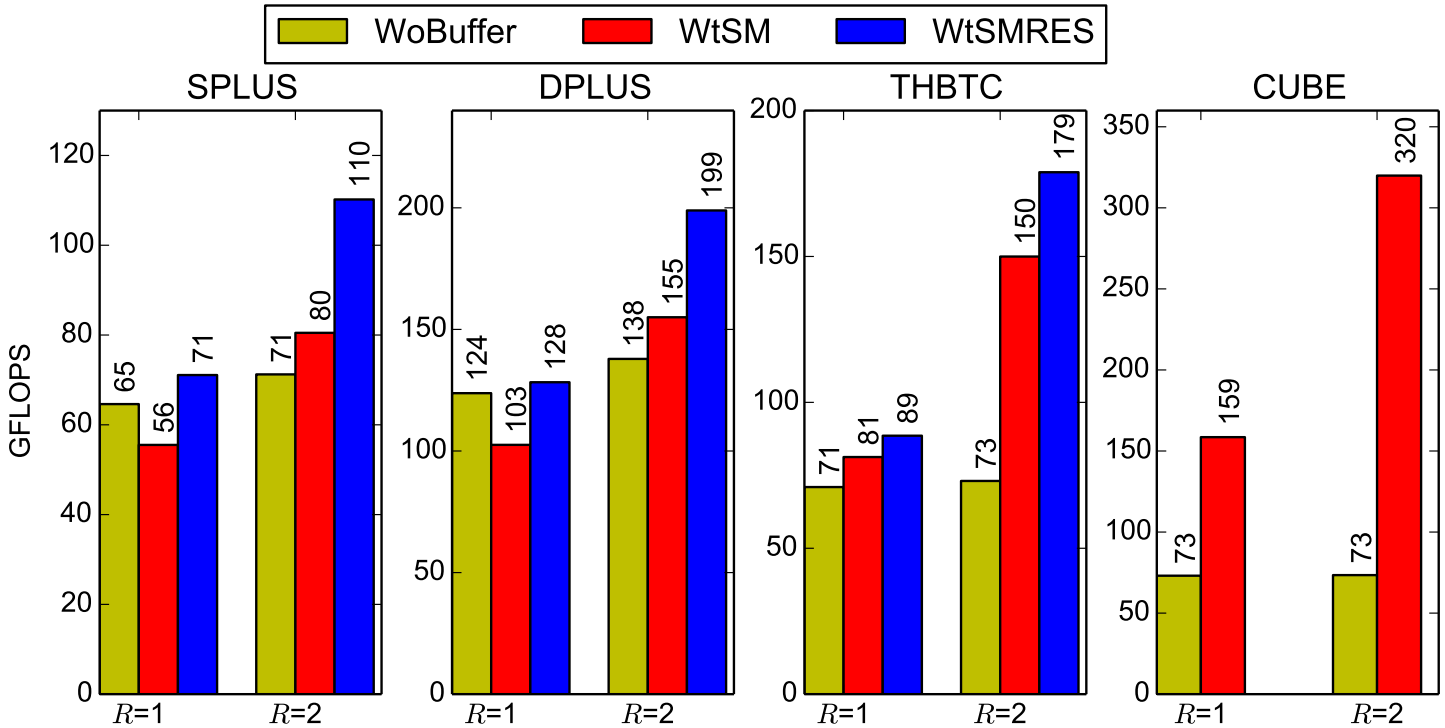
Performance difference between the predicted best and the measured best block configurations.

Overall performance prediction



- Measured (horizontal) and predicted (vertical) perf in GFLOPS.
- Each marker denotes the performance of a block configuration.

Best performance



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

Conclusion

- Execution driven auto-tuning puts tight limits on the size of the parameters space that can be explored
- Proposed model driven auto-tuning avoids the delay of execution driven model execution
- 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

Thanks!