Model-Driven Auto-Tuning of Stencil Computations on GPUs

Yue Hu\textsuperscript{1,2}, David M. Koppelman\textsuperscript{1,2}, Steven R. Brandt\textsuperscript{1,3}, and Frank Löffler\textsuperscript{1}

\textsuperscript{1}Center for Computation and Technology, Louisiana State University
\textsuperscript{2}School of Electrical Engineering and Computer Science, Louisiana State University
\textsuperscript{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 associativity) first
  - We are working on related micro-benchmarks for the measurements
Stencil code auto-generation

Memory hierarchy 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 configuration \((d_x, d_y, d_z, N_{\text{iter}}, \text{dir}_{\text{iter}})\) 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
  - \(N_{\text{iter}}\) denotes the number of stencil points each thread computes
  - \(\text{dir}_{\text{iter}}\) 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

\[ T_{\text{total}} = \sum_{i=0}^{7} N_{C_{\text{Block}_i}} \times T_{C_{\text{Block}_i}} \]

\[ \frac{1}{N_{\text{SM}} \times N_{\text{block/sm}}} \] (1)

- For a user provided block configuration \( C_{\text{Block}} \), we consider the non-partial \( (C_{\text{Block}_0}) \), plus all 7 possible partial executions \( (C_{\text{Block}_i}) \), for \( i = 1, \ldots, 7 \) during execution. A partial execution happens when some threads of a thread block are idle.
- \( N_{C_{\text{Block}_i}} \): the number of type \( i \) block configurations.
- \( T_{C_{\text{Block}_i}} \): execution time of a block with type \( i \) block configuration.
- \( N_{\text{block/sm}} \): the number of threads blocks per SM.
- \( N_{\text{SM}} \): the number of SMs the GPU has.
Modeling of a single thread block

\[ T(C_{\text{Block}}) = T_{\text{init}} + T_{\text{iter.1st}} + (N_{\text{iter}} - 1)T_{\text{iter.oth}} + T_{\text{diff}} \quad (2) \]

- 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 $T_{\text{iter}}$

\[
T_{\text{iter}} = T_{\text{gld}} + T_{\text{sst}} + T_{\text{sync}} + T_{\text{sld}} + T_{\text{comp}} + T_{\text{gst}}
\]

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

$$T_{\text{gld}} = \left[ \sum_{i=1}^{p} \frac{N^{(i)}_{\text{mem}}}{\lambda} \right] \times T_{\text{gmem-latency}} + \sum_{i=1}^{p} \left[ \frac{N^{(i)}_{\text{thd}} N^{(i)}_{\text{mem}}}{\theta_{\text{gmem/sm}} \beta_{\text{gm}}} \right]$$

- $p$: the number of terms needed for the corresponding computation
- $N^{(i)}_{\text{mem}}$: the number of data elements each thread will load
- $N^{(i)}_{\text{thd}}$: the number of threads assigned for such loading
- $\lambda$: the maximum number of elements that can be loaded per thread in parallel
- $\theta_{\text{gmem/sm}}$: the global memory throughput per SM
- $\beta_{\text{gm}}$: 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

In addition, we modeled:

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

See paper for details.
The model contains explicit and implicit GPU-dependent parameters. The value of explicit parameters, such as the number of double precision floating point operations per cycle (i.e. $\theta_{fp} = 64$), 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**

- **a) Single Plus (SPLUS)**
- **b) Double Plus (DPLUS)**
- **c) Thumbtack (THBTC)**
- **d) CUBE (CUBE)**

<table>
<thead>
<tr>
<th></th>
<th>SPLUS</th>
<th>DPLUS</th>
<th>THBTC</th>
<th>CUBE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$N_{\text{elem}}$</td>
<td>$1 + 6R$</td>
<td>$1 + 6R$</td>
<td>$(1 + 2R)^2 + 2R$</td>
<td>$(1 + 2R)^3$</td>
</tr>
<tr>
<td>$N_{\text{flop}}$</td>
<td>$1 + 6R$</td>
<td>$2 \times (1 + 6R)$</td>
<td>$(1 + 2R)^2 + 2R$</td>
<td>$(1 + 2R)^3$</td>
</tr>
</tbody>
</table>

- $R$: the radius of the stencil.
- $N_{\text{elem}}$ and $N_{\text{flop}}$: # 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 of auto-generated stencil code achieved by pre-running 200 random block configurations and selecting the best one.
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!