Enabling Automatic Partitioning of Data-Parallel Kernels with Polyhedral Compilation

Alexander Matz, Holger Fröning
Heidelberg University, Germany

LLVM Performance Workshop @CGO 2018
Sat 24 Feb 2018, Vienna, Austria
Multi GPU in the Real World

- NVIDIA DGX-1 and HGX-1
  8 Tesla GPUs

- Amazon AWS P2
  up to 16 Tesla GPUs

- Google Cloud Platform
  up to 8 Tesla GPUs

---

P2-Instance-Details

<table>
<thead>
<tr>
<th>Name</th>
<th>GPUs</th>
<th>vCPUs</th>
<th>RAM (GiB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2.xlarge</td>
<td>1</td>
<td>4</td>
<td>61</td>
</tr>
<tr>
<td>p2.8xlarge</td>
<td>8</td>
<td>32</td>
<td>488</td>
</tr>
<tr>
<td>p2.16xlarge</td>
<td>16</td>
<td>64</td>
<td>732</td>
</tr>
</tbody>
</table>

Observations GPU Programming

• Execution model
  • No guarantees exist for interactions among CTAs until kernel completion
    => Kernels can be safely partitioned along CTA boundaries (usually)

• Memory
  • Strong NUMA effects prohibit latency tolerance for remote accesses
  • Good partitioning mainly depends on memory access pattern

• Language
  • Data-parallel languages help in identifying areas of interest (kernels)
  • Parallel slackness helps for scalability (larger core count due to multi-GPU)
Basic Idea

- Keep clear data ownership and movements of single GPU programming
- Automatically sync buffers
- Hybrid compile time / run time approach
- Minimize runtime overhead
Pipeline Overview

- Based on LLVM (gpucc)
- Preprocessing based on text substitution
- Majority of functionality implemented as passes
- Not fully integrated yet
Kernel Analysis & Code Generation

1. Kernel Code

\[
\begin{align*}
    b[y*N + x] & = a[y*N + x]; \\
    b[y*N + x] & = a[y*N + x + 1]; \\
    b[y*N + x] & = a[y*N + x - 1]; \\
    b[y*N + x] & = a[y*N + x + N]; \\
    b[y*N + x] & = a[y*N + x - N]; \\
    b[y*N + x] & *= 0.2;
\end{align*}
\]

2. Application Model

Polyhedral Analysis

GPU Thread Grid

Array Access

3. Memory Range

Polyhedral Code Generation
Kernel Analysis

• Based on Polyhedral Value & Memory Analysis [1]

• Model should intuitively map Global ID → Array Element, so $\mathbb{Z}^3 \mapsto \mathbb{Z}^d$

• CUDA Expression “threadIdx + blockIdx * blockDim” not affine

• Workaround
  • Replace product with new input dimension “blockOffset”
  • Limit “threadIdx” to [0..“blockDim”], then project out

• Model is now: $\mathbb{Z}^6 \mapsto \mathbb{Z}^d$, with three pairs of two dependent dimensions

Code Generation

- Purpose A: Encode buffer dimension sizes and type information
- Purpose B: Implement efficient iterators for array accesses
  - Tracking buffer state requires iterators for write accesses
  - Synchronizing buffers for kernels requires iterators for read accesses
Iterator Code Generation

2D 5-point stencil, read map

\[
[N] \rightarrow \{
  I[y, x] \rightarrow S[o1=y, o2=x] : 0 <= o1, o2 < N;
  I[y, x] \rightarrow S[o1=y, o2=x-1] : 0 <= o1, o2 < N;
  I[y, x] \rightarrow S[o1=y, o2=x+1] : 0 <= o1, o2 < N;
  I[y, x] \rightarrow S[o1=y-1, o2=x] : 0 <= o1, o2 < N;
  I[y, x] \rightarrow S[o1=y+1, o2=x] : 0 <= o1, o2 < N;
\}

\[
[yl, yu, xl, xu] \rightarrow \{
  I[y, x] : 0 <= y <= yu and 0 <= xl <= x <= xu
\}

Identity schedule of map range

\[
\text{for (int } c0 = \max(\max(0, yl - 1), yl + xl - N); \]
  \text{c0 <= min(min(yu, N - 1), yu - xl + N - 1);}
\]
\[
\text{c0 += 1)}
\]
\[
\text{for (int } c1 = \max(\max(0, xl - 1), yl + xl - c0 - 1), -yu + xl + c0); \]
  \text{c1 <= min(min(xu, N - 1), yu + xu - c0 - 1), -yl + xu + c0);}
\]
\[
\text{c1 += 1)}
\]
\[
S(c0, c1);
\]

2D domain

\[
[yl, yu, xl, xu] \rightarrow \{
  I[y, x] : 0 <= y <= yu and 0 <= xl <= x <= xu
\}

- Based on isl AST generation
- Accurate but inefficient
- Reads don’t need 100% accuracy
- Last dimension is stored contiguous in memory in C
Iterator Code Generation

```c
for (int c0 = \text{max}(\text{max}(0, yl - 1), yl + xl - N); c0 <= \text{min}(\text{min}(yu, N - 1), yu - xl + N - 1); c0 += 1) {
    int y_lower = yl == c0 && yu >= c0 + 1 && xl == 0 && xu >= 2 ? 0 :
        c0 >= yl && yu >= c0 + 1 && xl >= 1 ? xl - 1 : xl;
    int y_upper = c0 >= yl && yu >= c0 + 1 && N >= xu + 2 ? xu :
        (yl == c0 + 1 && yu >= c0 + 2 && N >= xu + 1)
        || (c0 >= yl + 1 && yu == c0 && N >= xu + 1)
        || (yl >= c0 && yu >= c0 + 2 && xu == N) ? xu - 1 : N - 1;
    S(c0, y_lower, y_upper);
}
```

- Replace one loop with closed-form lower/upper expressions (optimized by LLVM)
- Good estimate for read maps
- Write maps need extra checks (modulo, non-convex sets) to verify accuracy
- Allows more efficient tracking and data transfers
Runtime Buffer Management

cudaMalloc(size) ->

foreach GPU:
Refs += [cudaMalloc(size)] ->
new Tracker()

cudaMemcpy(

foreach GPU:
maybeCopy (              )
update_tracker(              )

kernel<<<grid>>>(         )

foreach GPU: calc_partition()
foreach GPU: sync_buffer(              )
foreach GPU: kernel<<<partition>>>(              )
foreach GPU: update_tracker(              )
Runtime Buffer Synchronization

**First Kernel Launch**
- Data is in host memory
- Each GPU transfers its whole read set

**Kernel Iteration**
- Data is distributed on GPUs
- Each GPU only transfers stale data
- Often the most repeated part of application

**Data Gathering**
- Data is distributed on GPUs
- Host transfers most up to data chunk from each GPU
Runtime Buffer Tracking

• Synchronization requires tracking
• Track intervals of memory describing location of most recent update
• No overlapping intervals, implemented as b-tree based map with lower bound search
• Coalescing neighboring intervals keeps memory footprint and performance stable
Performance

Matrix Multiply

Speedup vs. Matrix side length

Hotspot (n = 32768)

Iterations vs. Speedup

N-Body (n = 262144)

Iterations vs. Speedup

Matrix Multiply (n = 28384)

Execution time (s) vs. GPUs

Hotspot (n = 28384, i = 1000)

Execution time (s) vs. GPUs

N-Body (n = 262144, i = 64)

Execution time (s) vs. GPUs
Future Work

• Fully integrated proof-of-concept

• Better handling of non-affine accesses

• More comprehensive validation using well-known benchmarks

• Array reshaping for better performance and memory utilization

• Explore shared memory optimizations (e.g. posted writes for synchronization)
Conclusion

• Compiler based Automatic Partitioning is feasible

• Polyhedral compilation is a good fit for GPU memory access patterns

• Accuracy of extracted memory access patterns crucial for both correctness (write accesses) and performance (read accesses)

• Performance of prototype experiments very promising

• LLVM provides excellent research platform for non-traditional compiler researchers
Thank you

We especially thank
Christoph Klein and Lorenz Braun (Heidelberg University), and
Johannes Doerfert (Saarland University) for their contributions to our research
as well as
Sudha Yalamanchili (Georgia Tech), Mark Hummel (NVIDIA), Peter Zaspel (University of Basel), Tobias Grosser
(ETH Zürich), Johannes Doerfert and Sebastian Hack (Saarland University) for many helpful discussions
and our Sponsors
BMBF, Google, NVIDIA, and the German Excellence Initiative
1D-Identity Map

1. Analysis Output

\[
\begin{align*}
[\text{boff}_x, \text{tid}_x] & \rightarrow \{ \\
[\text{ } ] & \rightarrow [\text{boff}_x + \text{tid}_x]
\}
\end{align*}
\]

2. 1D Iteration Domain (CUDA Thread Grid)

\[
\begin{align*}
[\text{boffmin}_x, \text{boffmax}_x, \text{bidmin}_x, \\
\text{bidmax}_x, \text{bdim}_x] & \rightarrow \{ \\
[\text{boff}_x, \text{bid}_x, \text{tid}_x] : \\
\text{boffmin}_x & \leq \text{boff}_x < \text{boffmax}_x \\
\text{and } \text{bidmin}_x & \leq \text{bid}_x < \text{bidmax}_x \\
\text{and } 0 & \leq \text{tid}_x < \text{bdim}_x
\}
\end{align*}
\]

3. Canonicalized Access Map

\[
\begin{align*}
[\text{boffmin}_x, \text{boffmax}_x, \text{bidmin}_x, \\
\text{bidmax}_x, \text{bdim}_x] & \rightarrow \{ \\
[\text{boff}_x, \text{bid}_x] & \rightarrow [00] : \\
\text{boffmin}_x & \leq \text{boff}_x < \text{boffmax}_x \\
\text{and } \text{bidmin}_x & \leq \text{bid}_x < \text{bidmax}_x \\
\text{and } \text{boff}_x & \leq 00 < \text{bdim}_x + \text{boff}_x
\}
\end{align*}
\]
2D 5-Point Stencil Read

Analysis Output

\[
\begin{align*}
[tid_x, boff_x, tid_y, boff_y, N] &\rightarrow \{ \\
[&] &\rightarrow A[o0, o1] : \\
& N > tid_x + boff_x \\
& \text{and } N > tid_y + boff_y \\
& \text{and } o0 \leq tid_y + boff_y \\
& \text{and } -1 + tid_x + boff_x + tid_y \\
& \text{+ } boff_y - o0 \leq o1 < N \\
& \text{and } o1 \leq 1 + tid_x + boff_x \\
& \text{+ tid}_y \text{ - boff}_y + o0; \\
[&] &\rightarrow A[1 + tid_y + boff_y, tid_x + boff_x] \\
\}
\end{align*}
\]

Canonicalized Access Map

\[
\begin{align*}
[bdim_y, bdim_x, boffmin_y, boffmax_y, boffmin_x, boffmax_x, \\
\text{bidmin}_y, \text{bidmax}_y, \text{bidmin}_x, \text{bidmax}_x, N] &\rightarrow \{ \\
[boff_y, boff_x] &\rightarrow A[o0, o1] : bdim_y = 1 \text{ and } bdim_x = 1 \\
& \text{and } \text{bidmax}_y > \text{bidmin}_y \text{ and } \text{bidmax}_x > \text{bidmin}_x \\
& \text{and } boffmin_y \leq boff_y < N \text{ and } boff_y < boffmax_y \\
& \text{and } boffmin_x \leq boff_x < N \text{ and } boff_x < boffmax_x \\
& \text{and } o0 \leq boff_y \text{ and } -1 + boff_y + boff_x - o0 \leq \\
& o1 \leq 1 - boff_y + boff_x + o0 \text{ and } o1 < N; \\
[boff_y, boff_x] &\rightarrow A[o0 = 1 + boff_y, o1 = boff_x] : \\
& bdim_y = 1 \text{ and } bdim_x = 1 \text{ and } \text{bidmax}_y > \text{bidmin}_y \\
& \text{and } \text{bidmax}_x > \text{bidmin}_x \\
& \text{and } boffmin_y \leq boff_y < boffmax_y \\
& \text{and } boffmin_x \leq boff_x < boffmax_x \\
\}
\end{align*}
\]