GPU-UNiCACHE: Automatic Code Generation of Spatial Blocking for Stencils on GPUs

Kaixi Hou, Hao Wang, Wu-chun Feng
{kaixihou,hwang121,wfeng}@vt.edu
Stencil Computations

- Nearest neighbor computations
  - Update every grid cell using its neighbors
  - Sweep over a structured grid (spatial dimension)
  - Iterate many times (time dimension)

Spatial Blocking for Stencil Computations

- High memory traffic + low arithmetic intensity
  - Memory bound computation
- Blocking optimizations are critical to achieve optimal performance
  - Different blocking strategies, e.g., 3D-blocking and 2.5D-blocking
Spatial Blocking for Stencil Computations

- High memory traffic + low arithmetic intensity
  - Memory bound computation
- Blocking optimizations are critical to achieve optimal performance
  - Different blocking strategies, e.g., 3D-blocking and 2.5D-blocking
  - Different GPU cache levels for the reusable data

A Compute Unit in GPUs *

Vector Registers (4x 64KB)  Local Data Share (64KB)  L1 Cache (16KB)

Performance

Programmability

* This figure serves as a logical diagram rather than a physical diagram
Motivation & Challenges

• Which cache level(s) should be selected?
  – Affected by different stencils, blocking strategies, platforms

**Different Stencils** with same platform and blocking strategy
Motivation & Challenges

• Which cache level(s) should be selected?
  – Affected by different stencils, blocking strategies, platforms
Motivation & Challenges

- Which cache level(s) should be selected?
  - Affected by different stencils, blocking strategies, platforms

**Different Stencils** with same platform and blocking strategy

**Different Platforms** with same stencil and blocking strategy

**Different Blocking Strategies** with same stencil and platform
Motivation & Challenges

- Which cache level(s) should be selected?
  - Affected by different stencils, blocking strategies, platforms

To search for the suitable cache level requires a thorough investigation of different stencil kernels optimized by different cache levels.

Different Stencils with same platform and blocking strategy

Different Platforms with same stencil and blocking strategy

Different Blocking Strategies with same stencil and platform
Talk Outline

• Introduction & Motivation

• Background
  – GPU Register Data Exchange

• GPU-UNICACHE Framework
  – Writing Stencils with GPU-UNICACHE
  – GPU-UNICACHE Framework
  – RegCache Method -- fetch()
  – Other Methods

• Evaluation & Discussion
• Conclusion
GPU Register Data Exchange

- Local/shared memory was introduced for efficient communication and data sharing between threads

- Modern GPUs support an even faster way (i.e., registers as cache)
  - Directly data exchange in thread registers

- Fastest memory for each thread
- No explicit synchronization
- However, …
GPU Register Data Exchange

• Using registers as cache in stencils is non-trivial
  – What are the communication patterns?

Data distribution (thread view)

When reading the cells of northwest (NW) neighbors, they need to first know the correct “friend” threads.
GPU Register Data Exchange

- Using registers as cache in stencils is non-trivial
  - What are the communication patterns?
    - When reading the cells of northwest (NW) neighbors, they need to first know the correct "friend" threads.
    
    Data distribution (thread view)
    
    - Which registers are of interest for exchange?
      - When reading the cells of the north (N) neighbors, they also need to find the correct registers in the correct "friend" threads.
Using registers as cache in stencils is non-trivial

- What are the communication patterns?

  When reading the cells of northwest (NW) neighbors, they need to first know the correct "friend" threads.

- Which registers are of interest for exchange?

  When reading the cells of the north (N) neighbors, they also need to find the correct registers in the correct "friend" threads.

- The answers are determined by specific stencils, execution unit sizes and dimensionalities (platforms).
• Using registers as cache in stencils is non-trivial
  – What are the communication patterns?

Therefore, we propose GPU-UNICACHE to ...
• *Handle* comp. & comm. patterns
• *Generate* different spatial blocking codes
• *Achieve* performance portability among GPUs

– The answers are determined by specific stencils, execution unit sizes and dimensionalities (platforms).
Talk Outline

• Introduction & Motivation

• Background
  – GPU Register Data Exchange

• GPU-UNICACHE Framework
  – Writing Stencils with GPU-UNICACHE
  – GPU-UNICACHE Framework
  – RegCache Method -- \textit{fetch()}
  – Other Methods

• Evaluation & Discussion

• Conclusion
Writing Stencils with GPU-UNICACHE API

• Cell-based library APIs
  – C functions describing scalar execution on one grid element
  – Executed over Cartesian domains
  – Users still have tight control of how to design stencil codes

• Interface of GPU-UNICACHE functions

```cpp
template<class T>
class GPU_UniCache {

protected:
  __device__ virtual T _load(int z, int y=0, int x=0) [[hc]];
  __device__ virtual void _store(T v, int z, int y=0, int x=0) [[hc]];

public:
  __device__ virtual void init(T *in, int off, int mode=CYCLIC) [[hc]];
  __device__ virtual T fetch(int z, int y, int x, int tc_i=0) [[hc]];
};

// Derived classes
class L1Cache : public GPU_UniCache{ ... };
class LDSCache: public GPU_UniCache{ ... };
class RegCache: public GPU_UniCache{ ... };
```

CUDA version

HCC version
Writing Stencils with GPU-UNiCACHE API

- Stencil kernel using GPU-UNiCACHE functions
  - E.g., 2D-5Point stencil kernel using registers as cache
  - The kernel codes are cross-platform

```c
__global__
void kern_2d5pt(float *in, float *out, float a0-4)
{
    RegCache<float> buf(m, n, h, 4*);

    buf.init(in, 0, CYCLIC);
    // Each thread processes 4 points since csr_fct = 4
    for (csr_id = 0; csr_id < 4; csr_id++)
        out[/*global idx w/ offset csr_id*/] =
            a0 * buf.fetch(-1, 0, csr_id) +
            a1 * buf.fetch(0, -1, csr_id) +
            a2 * buf.fetch(0, 0, csr_id) +
            a3 * buf.fetch(0, 1, csr_id) +
            a4 * buf.fetch(1, 0, csr_id);
}
```

* Codes are written with thread coarsening factor csr_fct of 4, which means each thread will update 4 cell points. The current cell point is located by using csr_id.
• Overview of the structure
RegCache Method – *fetch()*

- Need some inputs to tell what the stencil looks like

- `warp_dim[3] = {4,2,0};`
- `ghst_lyr = 1;`
- `warp_size = 8;`
- `sten_dim = 2;`
- `crs_fct = 1;`
- `crs_dim = 1;`

// dimensions of warp (exponents)
// how many ghost layers
// warp size, e.g. 64 (AMD), 32 (NV)
// stencil dimensions
// thread coarsening factor
// coarsening on which dim

2D stencil with the execution unit of 2x4 threads
RegCache Method – *fetch()*

- Use formalized construct codes to facilitate code generation

Formalized data exchange constructs (*Red* parameters are determined by specific stencils)

```plaintext
friend_idX = (lane_id+F1+((lane_id)>>warp_dim[0])*2*ghost_lyr) & (warp_size-1);
txX = data_exchange(regN1, friend_idX);
tyX = data_exchange(regN2, friend_idX);
tzX = data_exchange(regN3, friend_idX);
Return ((lane_id < M1)? txX: ((lane_id < M2)? tyX: tzX));
```
RegCache Method – *fetch(*)*

- Use formalized construct codes to facilitate code generation

Formalized data exchange constructs *(Red parameters are determined by specific stencils)*

```plaintext
friend_idX = (lane_id+F1+((lane_id>>>warp_dim[0])*2*ghst_lyr))&(warp_size-1);
txX = data_exchange(regN1, friend_idX);
tyX = data_exchange(regN2, friend_idX);
tzX = data_exchange(regN3, friend_idX);
Return ((lane_id < M1)? txX: ((lane_id < M2)? tyX: tzX));
```

```
<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
</tr>
</thead>
<tbody>
<tr>
<td>t6</td>
<td>t7</td>
<td>t0</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
</tr>
<tr>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
<td>t0</td>
<td>t1</td>
</tr>
<tr>
<td>t2</td>
<td>t3</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
</tr>
</tbody>
</table>
```
RegCache Method – \textit{fetch()}

- Use formalized construct codes to facilitate code generation

Formalized data exchange constructs (Red parameters are determined by specific stencils)

\[
\text{friend\_idX} = (\text{lane\_id} + \text{F1} + ((\text{lane\_id} \gg \text{warp\_dim[0]})) \times 2 \times \text{ghst\_lyr}) \& (\text{warp\_size} - 1);
\]
\[
\text{txX} = \text{data\_exchange(\text{regN1}, \text{friend\_idX});}
\]
\[
\text{tyX} = \text{data\_exchange(\text{regN2}, \text{friend\_idX});}
\]
\[
\text{tzX} = \text{data\_exchange(\text{regN3}, \text{friend\_idX});}
\]

\text{Return ((lane\_id < M1)? txX: ((lane\_id < M2)? tyX: tzX));}

Accessing NE neighbors

The NE neighbors start from tid 2 and reg [Red]
RegCache Method – *fetch()*

- Use formalized construct codes to facilitate code generation

Formalized data exchange constructs (Red parameters are determined by specific stencils)

```plaintext
friend_idX = (lane_id+F1+((lane_id>>warp_dim[0])*2*ghst_lyr))&(warp_size-1);
txX = data_exchange(regN1, friend_idX);
tyX = data_exchange(regN2, friend_idX);
tzX = data_exchange(regN3, friend_idX);
Return ((lane_id < M1)? txX: ((lane_id < M2)? tyX: tzX));
```

The NE neighbors start from tid 2 and reg

Accessing NE neighbors

Calculated parameters

```plaintext
friend_id1 = (lane_id+2+((lane_id>>2)*2*1))&(8-1);
tx1 = data_exchange(reg , friend_id1);
ty1 = data_exchange(reg , friend_id1);
return ((lane_id < 4)? tx1: ty1);
```
RegCache Method – \texttt{fetch()}

- Map data exchange instructions to real codes

**Calculated parameters**

```
friend_id1 = (lane_id+2+((lane_id>>2)*2*1))&(8-1);
tx1 = data_exchange(reg, friend_id1);
ty1 = data_exchange(reg, friend_id1);
return ((lane_id < 4)? tx1: ty1);
```

**CUDA codes**

```
friend_id = (lane_id+2+(lane_id>>2)*2)&7;
tx = _shfl(r, friend_id);
ty = _shfl(s, friend_id);
return lane_id < 4? tx: ty;
```

**HCC codes**

```
friend_id = (lane_id+2+(lane_id>>2)*2)&7;
tx = _amdgcn_ds_bpermute(friend_id<<2, r);
ty = _amdgcn_ds_bpermute(friend_id<<2, s);
return lane_id < 4? tx: ty;
```
• Increasing workloads per thread
• Changing compute domain dimensions

```
friend_id0 = (lane_id + 0) & 3;
tx0 = __shfl(reg0, friend_id0);
return tx0;
```

```
friend_id2 = (lane_id + 2) & 3;
tx2 = __shfl(reg6, friend_id2);
ty2 = __shfl(reg7, friend_id2);
return (lane_id < 2) ? tx2 : ty2;
```

```
frend_id0 =
(lane_id+0+((lane_id>>1) *2)) & 3;
tx0 = __shfl(reg0, friend_id0);
ty0 = __shfl(reg1, friend_id0);
return (lane_id < 2) ? tx0 : ty0;
```

```
frend_id2 =
(lane_id+2+((lane_id>>1) *2)) & 3;
tx2 = __shfl(reg6, friend_id2);
ty2 = __shfl(reg7, friend_id2);
return (lane_id < 2) ? tx2 : ty2;
```
Other Methods

- Support hardware-managed L1 cache
- Support explicit blocking via local/shared memory
  - Branch-style: threads on boundary load more data
  - Cyclic-style: threads load data in a round-robin fashion

Loading Branch-Style

```
<table>
<thead>
<tr>
<th>t0</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t3</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>t0</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
<td>t3</td>
</tr>
<tr>
<td>t4</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
<td>t7</td>
</tr>
<tr>
<td>t4</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
<td>t7</td>
</tr>
</tbody>
</table>
```

Loading Cyclic-Style

```
<table>
<thead>
<tr>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
</tr>
</thead>
<tbody>
<tr>
<td>t6</td>
<td>t7</td>
<td>t0</td>
<td>t1</td>
<td>t2</td>
<td>t3</td>
</tr>
<tr>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
<td>t0</td>
<td>t1</td>
</tr>
<tr>
<td>t2</td>
<td>t3</td>
<td>t4</td>
<td>t5</td>
<td>t6</td>
<td>t7</td>
</tr>
</tbody>
</table>
```
Talk Outline

• Introduction & Motivation

• Background
  – GPU Register Data Exchange

• GPU-UNICACHE Framework
  – Writing Stencils with GPU-UNICACHE
  – GPU-UNICACHE Framework
  – RegCache Method -- fetch()
  – Other Methods

• Evaluation & Discussion

• Conclusion
## Experiment Setups

### Experiment Testbeds

<table>
<thead>
<tr>
<th></th>
<th>AMD</th>
<th>NVIDIA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Model</strong></td>
<td>Radeon R9 Nano</td>
<td>GeForce GTX 980</td>
</tr>
<tr>
<td><strong>Codename</strong></td>
<td>Fiji XT</td>
<td>GM204(Maxwell)</td>
</tr>
<tr>
<td><strong>Cores</strong></td>
<td>4096</td>
<td>2048</td>
</tr>
<tr>
<td><strong>Core frequency</strong></td>
<td>1000 MHz</td>
<td>1126 MHz</td>
</tr>
<tr>
<td><strong>Register file size</strong></td>
<td>256 KB*</td>
<td>256 KB</td>
</tr>
<tr>
<td><strong>L1/LDS/L2</strong></td>
<td>16/64/1024 KB</td>
<td>-/96/2048 KB</td>
</tr>
<tr>
<td><strong>Memory bus</strong></td>
<td>HBM</td>
<td>GDDR5</td>
</tr>
<tr>
<td><strong>Memory capacity</strong></td>
<td>4096 MB</td>
<td>4096 MB</td>
</tr>
<tr>
<td><strong>Memory BW</strong></td>
<td>512 GB/s</td>
<td>224 GB/s</td>
</tr>
<tr>
<td><strong>GFLOPS flt/dbl</strong></td>
<td>8192/512</td>
<td>4612/144</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>HCC/ROCM_1.2</td>
<td>CUDA_7.5</td>
</tr>
</tbody>
</table>

* Each CU has 256 KB vector registers and additional 8 KB scalar registers.
Experiment Setups

- List of Benchmark Stencil Kernels (w/ Float data type)

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Default/L1</th>
<th>LDS</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernels</td>
<td>Blk Strat.</td>
<td>Blk Strat.</td>
<td>Variants</td>
</tr>
<tr>
<td>3D Stencils</td>
<td>• 2.5D-Blk</td>
<td>• 2.5D-Blk</td>
<td>• 2.5D-Blk</td>
</tr>
<tr>
<td></td>
<td>• 3D-Blk</td>
<td>• 3D-Blk</td>
<td>• 3D-Blk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Branch</td>
<td>• Branch</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Cyclic</td>
<td>• Cyclic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 1D-WF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• 2D-WF</td>
</tr>
</tbody>
</table>

- We want to test different combinations of stencils, blocking strategies, and compute domains.

- Performance Metrics: \( GFLOPS = \frac{NUM\_OPS \times x \times y \times z}{time} \)

“Please read our paper and see more experiments of
- 1D 2D stencils
- Double data type”
Evaluation

- Use different GPU-UNICACHE objects on AMD GPU

- 2.5D blocking is preferred in 3D stencils

3D-7Point

3D-27Point
Evaluation

- Use different GPU-UNiCACHE objects on AMD GPU

- 2.5D blocking is preferred in 3D stencils
- RegCache codes can achieve better performance than using LDSCache especially in 3D-27Point
Evaluation

- Use different GPU-UNICACHE objects on NVIDIA GPU

- Different GPU-UNICACHE codes are more sensitive to workloads per thread
Comparison with Open-Source Benchmarks

- Compare to the stencils with only spatial blocking
- Choose our best performant GPU-UNICACHE codes

- By simply using the GPU-UNICACHE functions, we can outperform the existing benchmarks by up to 1.5x
Conclusion

• Propose a framework to automatically generate cell-based library codes to use different cache levels for stencils computations.
  – Support different stencils
  – Support different blocking strategies
  – Support different platforms

• Performance:
  – Evaluate relationships between different stencils and cache levels
  – Up to 1.5x performance benefit over existing stencil benchmarks

THANK YOU! More info: http://synergy.cs.vt.edu