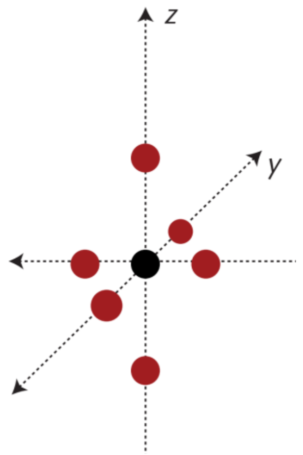
GPU-UniCache: Automatic Code Generation of Spatial Blocking for Stencils on GPUs

Kaixi Hou, Hao Wang, **Wu-chun Feng**

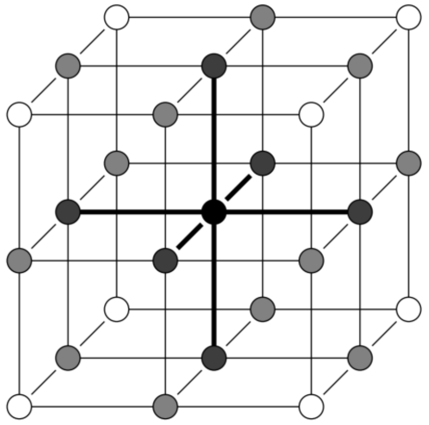
{kaixihou,hwang121,wfeng}@vt.edu

Stencil Computations

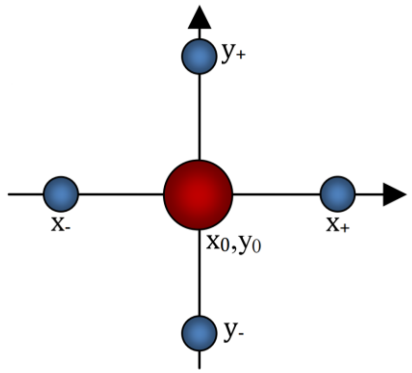
- Nearest neighbor computations
  - Update every grid cell using its neighbors
  - Sweep over a structured grid (**spatial dimension**)
  - Iterate many times (time dimension)



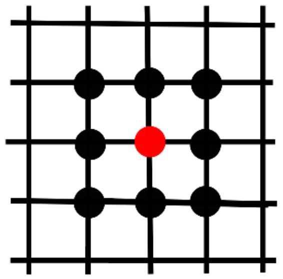
3D Stencil
(7 Points)[1]



3D Stencil
(27 Points)[2]



2D Stencil
(5 Points)[3]

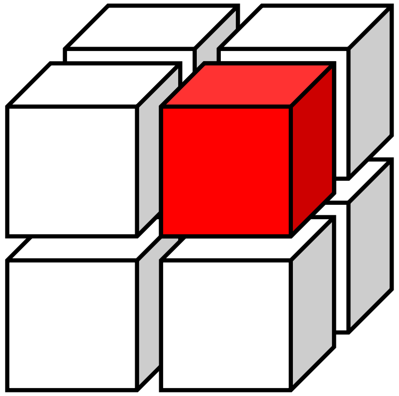


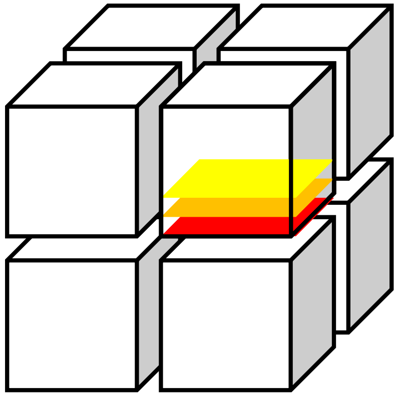
2D Stencil
(9 Points)[4]

[1] X. Cai, *et al.* "Accelerating a 3D Finite-Difference Earthquake Simulation with a C-to-CUDA Translator", IEEE CS&E (2012).
[2] F. Mueller, *et al.* "Autogeneration and Autotuning of 3D Stencil Codes on Homogeneous and Heterogeneous GPU Clusters", IEEE TPDS (2013).
[3] M. J. Gourlay, "Fluid Simulation for Video Games (part 6)“. Intel online articles (2012).
[4] Compact stencil, https://en.wikipedia.org/wiki/Compact_stencil

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

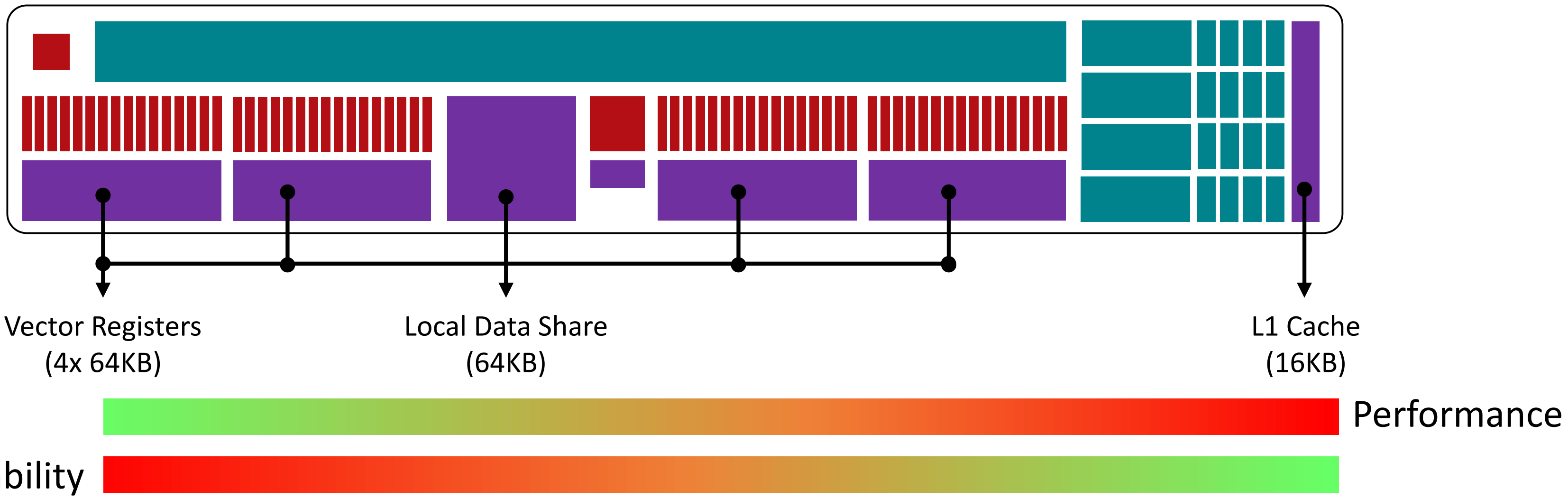

3D-blocking


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

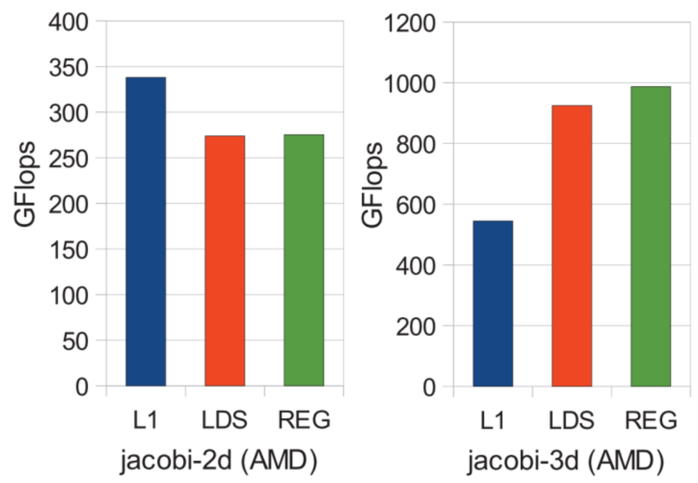
A Compute Unit in GPUs *



* This figure serves as a logical diagram rather than a physical diagram

Motivation & Challenges

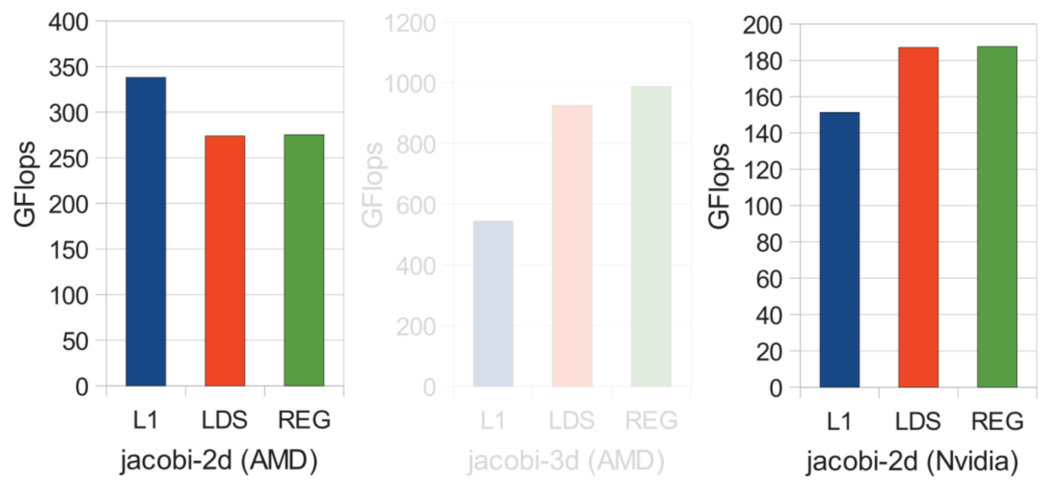
- Which cache level(s) should be selected?
  - Affected by different stencils, blocking strategies, platforms



Different Stencils with same platform and blocking strategy

Motivation & Challenges

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

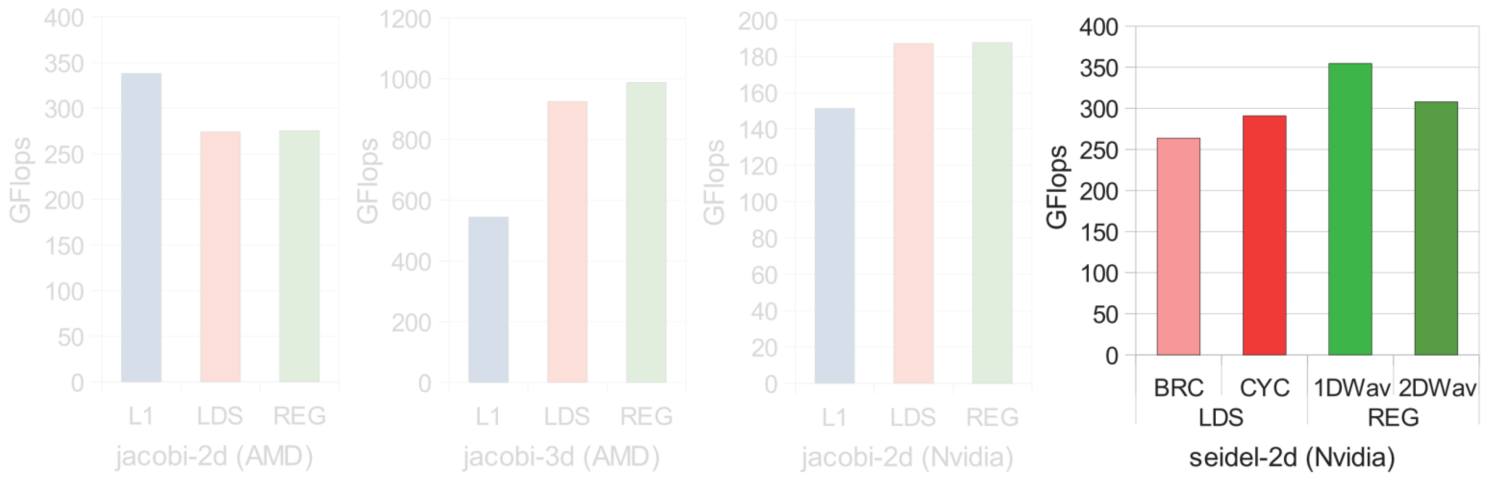


Different Stencils with same platform and blocking strategy

Different Platforms with same stencil and blocking strategy

Motivation & Challenges

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



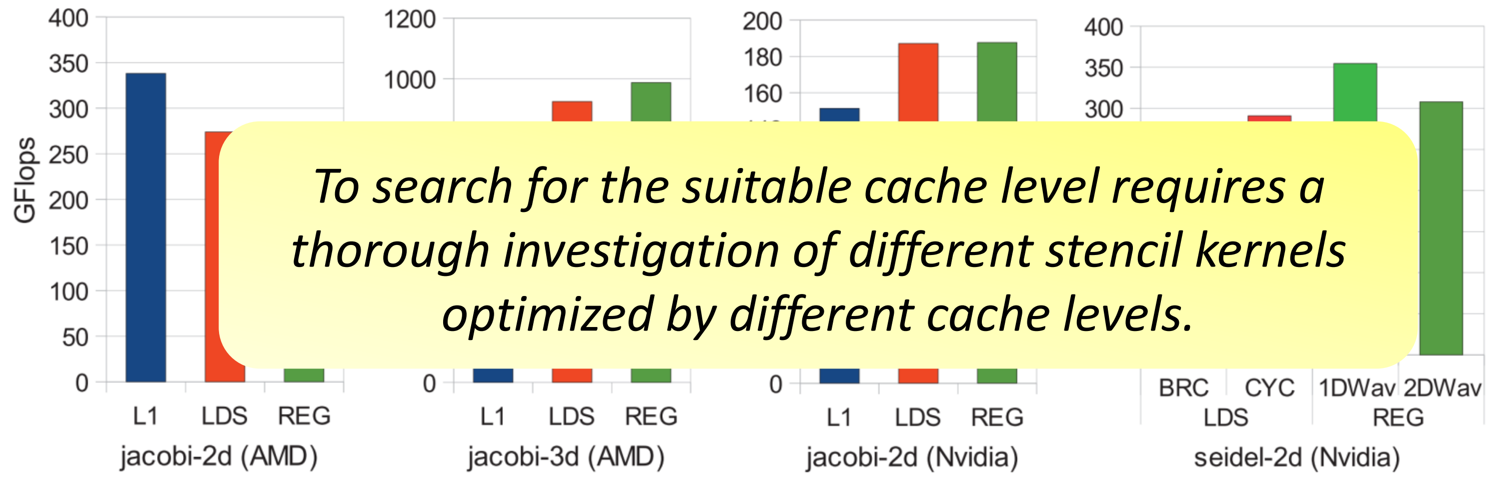
Different Stencils with same platform and blocking strategy

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.

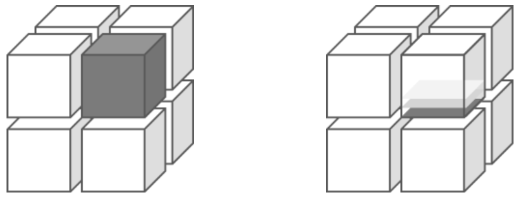
Different Stencils with same platform and blocking strategy

Different Platforms with same stencil and blocking strategy

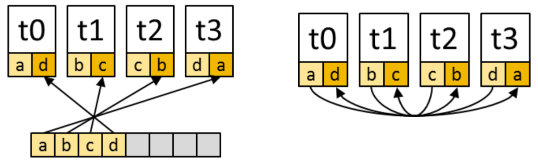
Different Blocking Strategies with same stencil and platform

Talk Outline

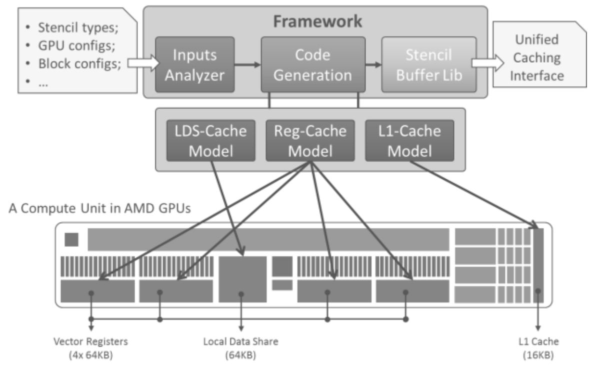
- Introduction & Motivation



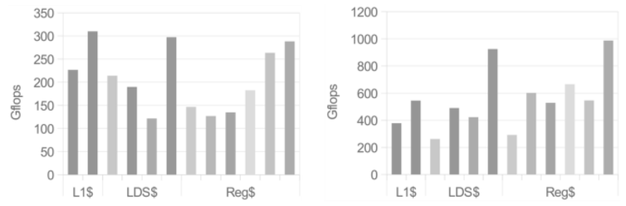
- Background
  - GPU Register Data Exchange



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

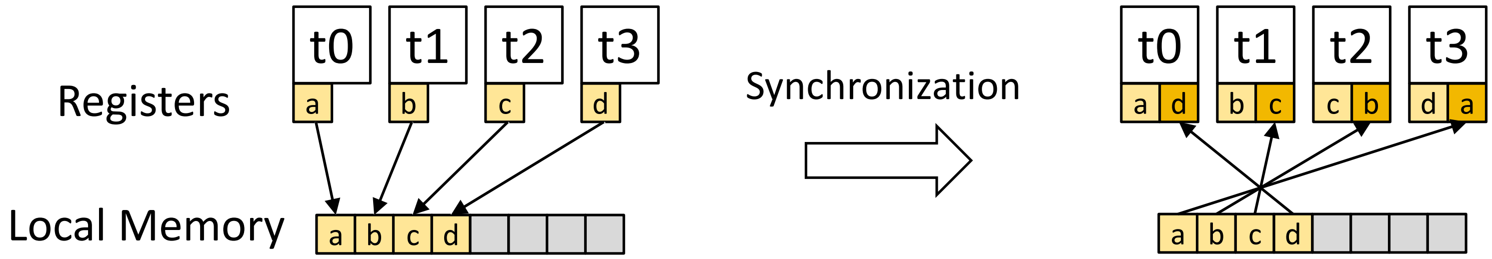


- Evaluation & Discussion
- Conclusion

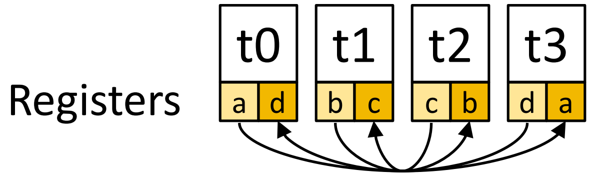


GPU Register Data Exchange

- 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

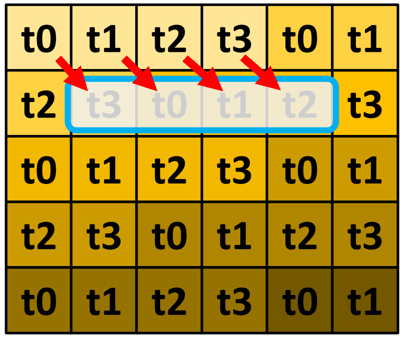


- *Fastest memory for each thread*
- *No explicit synchronization*
- **However, …**

GPU Register Data Exchange

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

Data distribution (thread view)

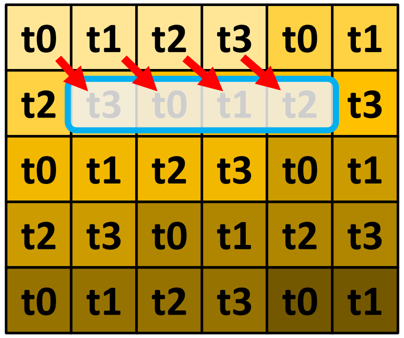


When reading the cells of northwest (NW) neighbors, they need to first know ***the correct "friend" threads***.

GPU Register Data Exchange

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

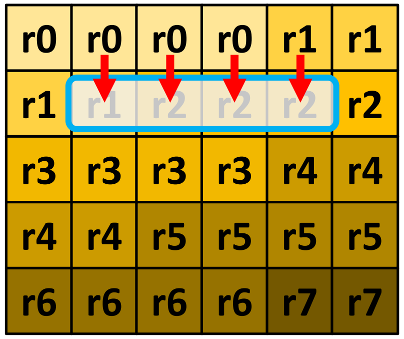
Data distribution
(thread view)



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

  - Which registers are of interest for exchange?

Data distribution
(register view)

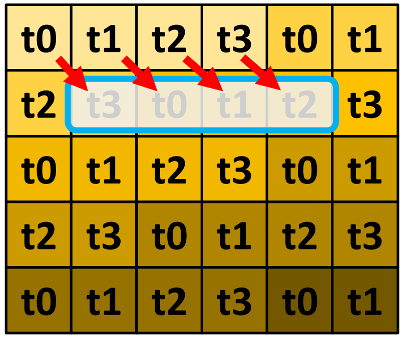


When reading the cells of the north (N) neighbors, they also need to find ***the correct registers*** in the correct "friend" threads.

GPU Register Data Exchange

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

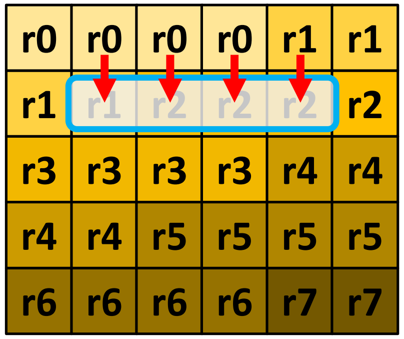
Data distribution (thread view)



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

  - Which registers are of interest for exchange?

Data distribution (register view)



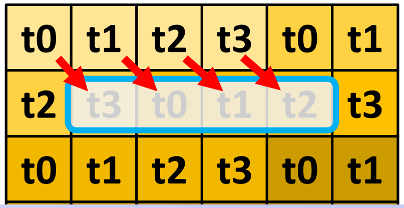
When reading the cells of the north (N) neighbors, they also need to find ***the correct registers*** in the correct "friend" threads.

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

GPU Register Data Exchange

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

Data distribution (thread view)

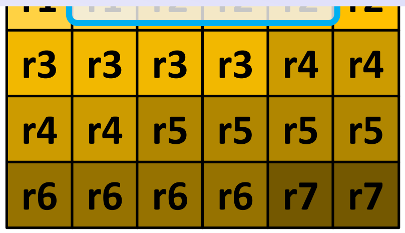


When reading the cells of northwest (NW) neighbors, they need to first know ***the correct “friend” threads.***

Therefore, we propose GPU-UniCache to …

- *Handle* comp. & comm. patterns
- *Generate* different spatial blocking codes
- *Achieve* performance portability among GPUs

Data distribution (register view)

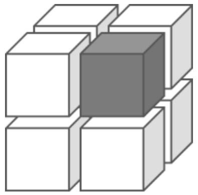
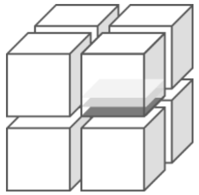


When reading the cells of the north (N) neighbors, they also need to find ***the correct registers*** in the correct “friend” threads.

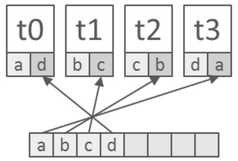
- The answers are determined by specific stencils, execution unit sizes and dimensionalities (platforms).

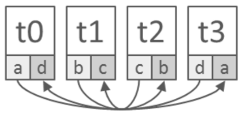
Talk Outline

- Introduction & Motivation

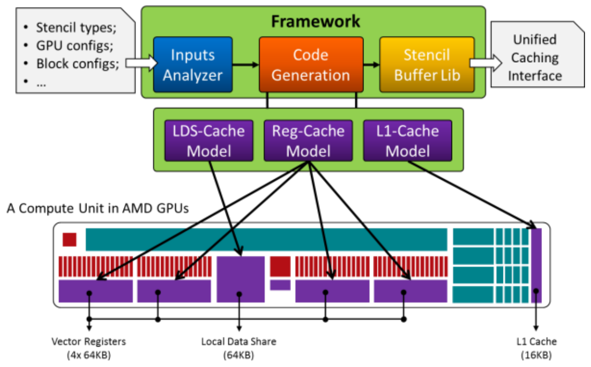



- Background
  - GPU Register Data Exchange

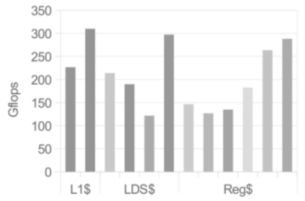


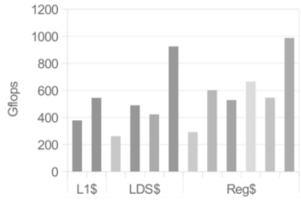


- GPU-UNICACHE Framework
  - Writing Stencils with GPU-UNICACHE
  - GPU-UNICACHE Framework
  - RegCache Method -- *fetch()*
  - Other Methods



- Evaluation & Discussion
- Conclusion





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

```
        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

```
__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);
}
```

Instantiating RegCache obj.

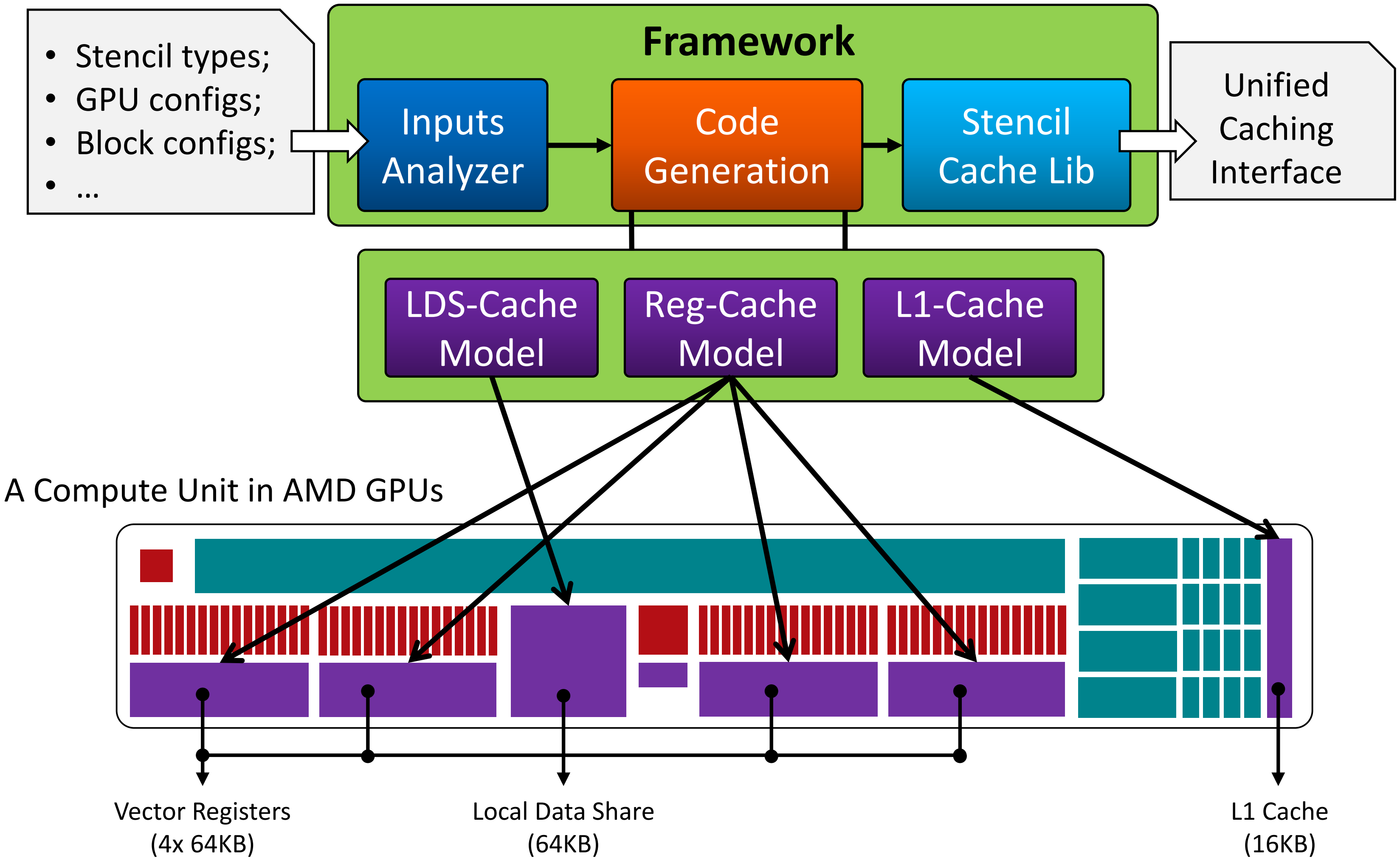
Loading data to RegCache obj.

Fetching data from RegCache obj.

* 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*.

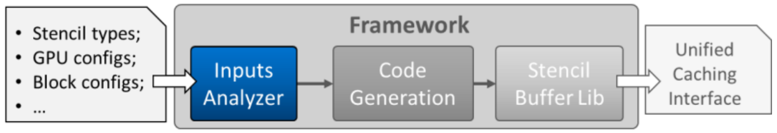
GPU-UniCache Framework

- Overview of the structure

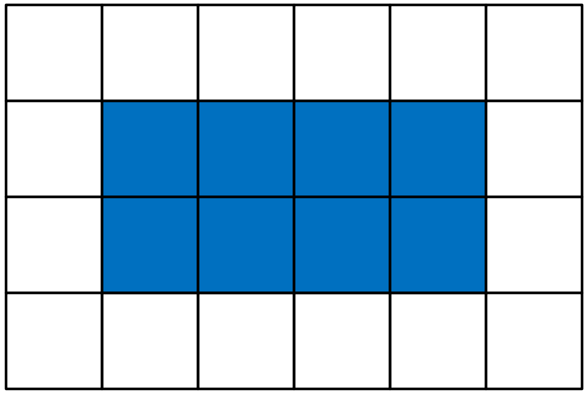


RegCache Method – *fetch()*

- Need some inputs to tell what the stencil looks like



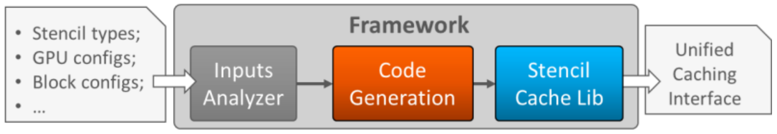
```
• warp_dim[3] = {4,2,0};   // dimensions of warp (exponents)
• ghst_lyr = 1;            // how many ghost layers
• warp_size = 8;           // warp size, e.g. 64 (AMD), 32 (NV)
• sten_dim = 2;            // stencil dimensions
• crs_fct = 1;             // thread coarsening factor
• crs_dim = 1;             // coarsening on which dim
```



2D stencil with the execution unit of 2x4 threads

RegCache Method – *fetch()*

- Use formalized construct codes to facilitate code generation



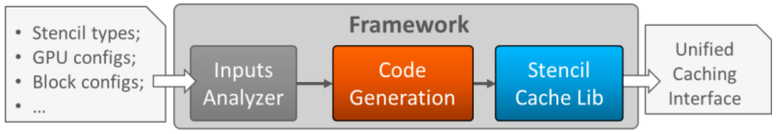
Formalized data exchange constructs (Red parameters are determined by specific stencils)

```
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));
```

t0	t1	t2	t3	t4	t5
t6	t7	t0	t1	t2	t3
t4	t5	t6	t7	t0	t1
t2	t3	t4	t5	t6	t7

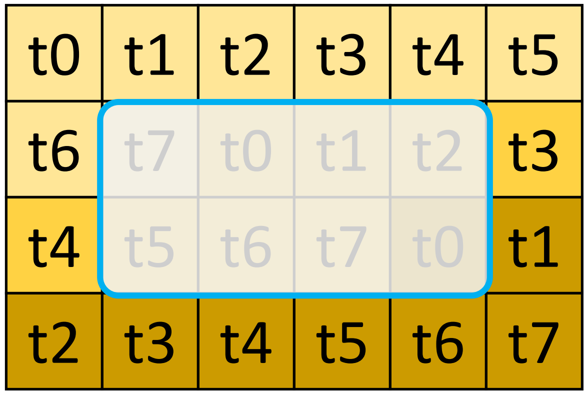
RegCache Method – *fetch()*

- Use formalized construct codes to facilitate code generation



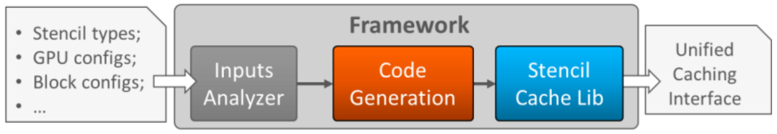
Formalized data exchange constructs (Red parameters are determined by specific stencils)

```
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));
```



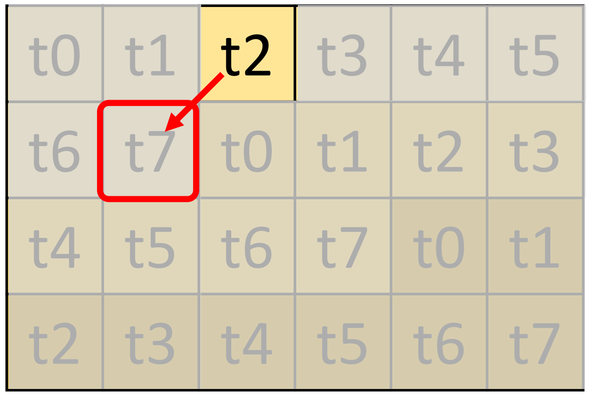
RegCache Method – *fetch()*

- Use formalized construct codes to facilitate code generation



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

```
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));
```

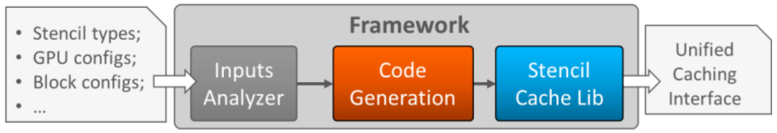


Accessing NE neighbors

The NE neighbors start from **tid 2** and **reg** 🟨

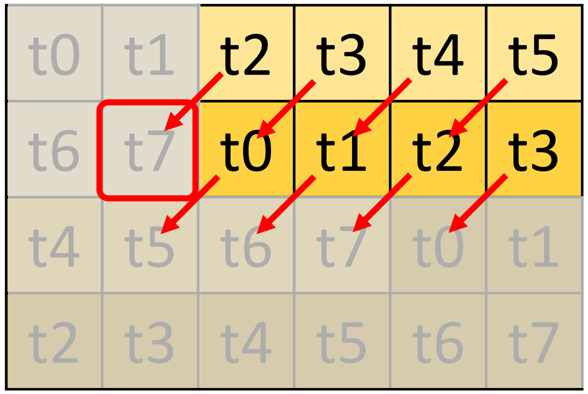
RegCache Method – *fetch()*

- Use formalized construct codes to facilitate code generation



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

```
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));
```



Accessing NE neighbors

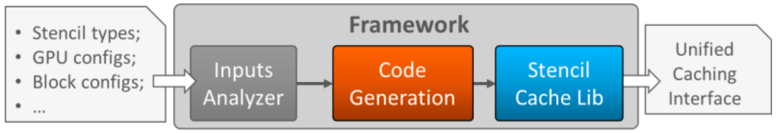
The NE neighbors start from **tid 2** and **reg** 🟨

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);
```

RegCache Method – *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

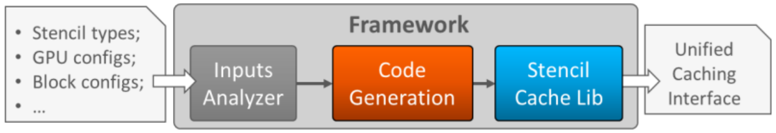
```
friend_id = (lane_id+2+(lane_id>>2)*2)&7;
tx = _shfl(r, friend_id);
ty = _shfl(s, friend_id);
return lane_id < 4? tx: ty;
```

HCC codes

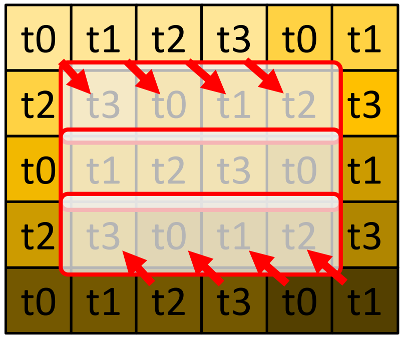
```
friend_id = (lane_id+2+(lane_id>>2)*2)&7;
tx = _amdgcn_ds_bpermute(friend_id<<2, r);
ty = _amdgcn_ds_bpermute(friend_id<<2, s);
return lane_id < 4? tx: ty;
```

RegCache Method – *fetch()*

- Increasing workloads per thread
- Changing compute domain dimensions



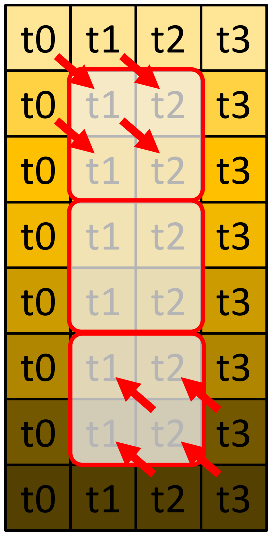
```
friend_id0 = (lane_id + 0) & 3;
tx0 = __shfl(reg0, friend_id0);
return tx0;
```



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

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

```
friend_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;
```

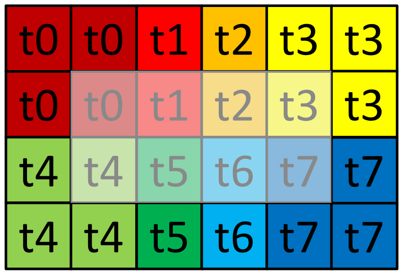


- warp_dim[3] = {2,2,0};
- crs_fct = 3;
- crs_dim = 1;
- ghst_lyr = 1;
- warp_size = 4;
- sten_dim = 2;

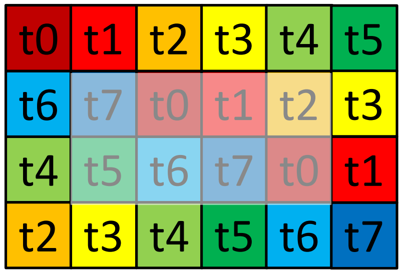
```
friend_id2 =
(lane_id+2+((lane_id>>1)*2))&3 ;
tx2 = __shfl(reg6, friend_id2);
ty2 = __shfl(reg7, friend_id2);
return (lane_id < 2 )? tx2: ty2;
```

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



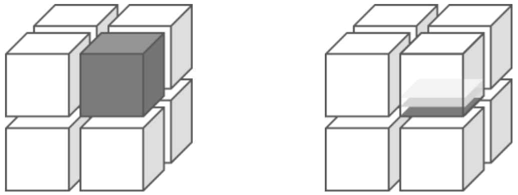
Loading Branch-Style



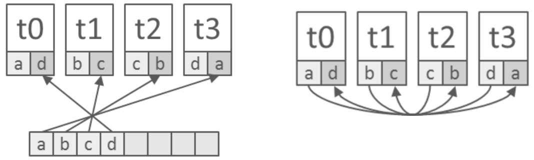
Loading Cyclic-Style

Talk Outline

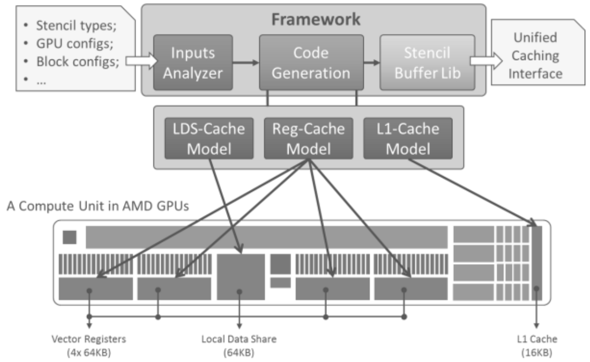
- Introduction & Motivation



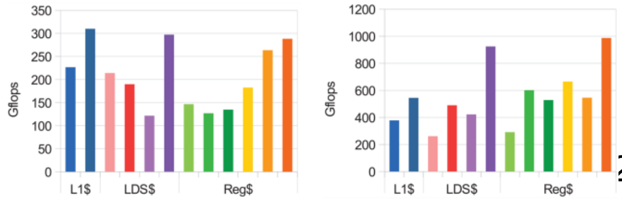
- Background
  - GPU Register Data Exchange



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



- **Evaluation & Discussion**
- Conclusion



Experiment Setups

- Experiment Testbeds

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

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

Experiment Setups

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

Buffer	Default/L1	LDS		Register	
Kernels	Blk Strat.	Blk Strat.	Variants	Blk Strat.	Variants
3D Stencils (3D-7Pt/3D-27Pt)	• 2.5D-Blk • 3D-Blk	• 2.5D-Blk • 3D-Blk	• Branch • Cyclic	• 2.5D-Blk • 3D-Blk	• 1D-WF • 2D-WF

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

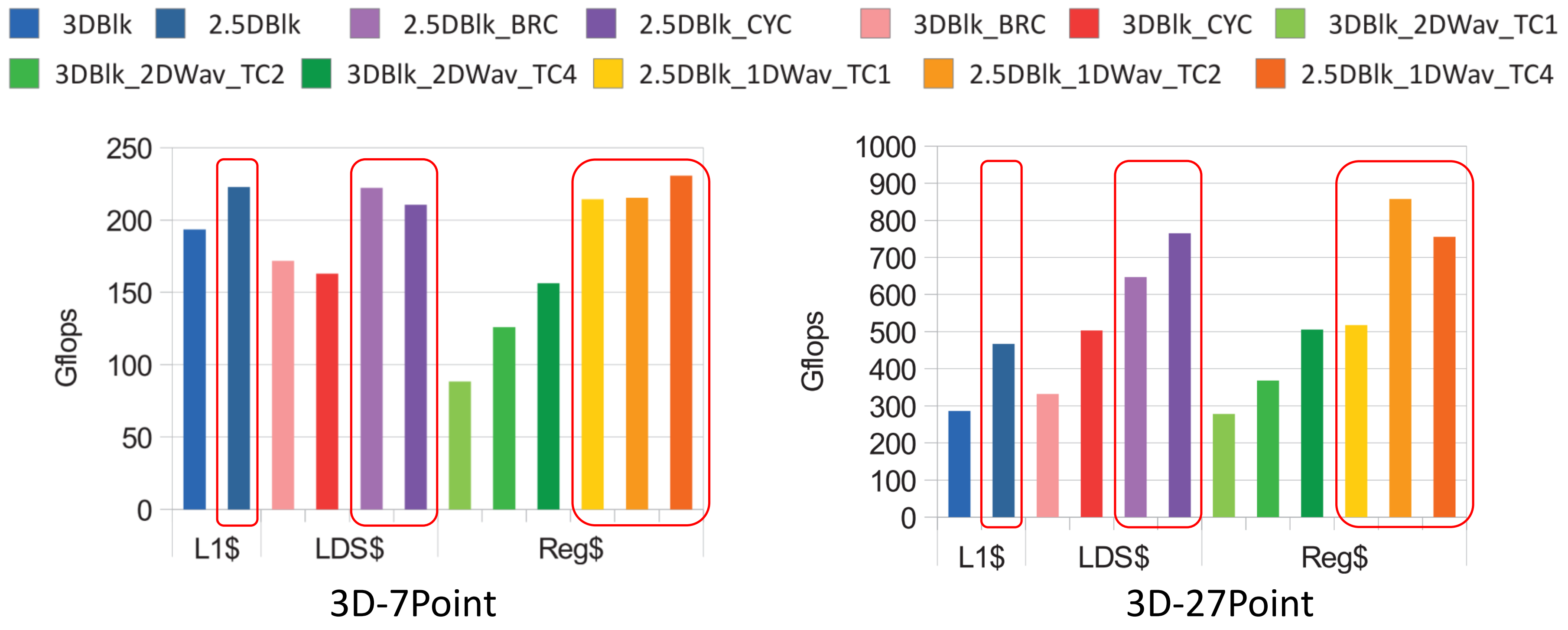
- Performance Metrics: $\boldsymbol{GFLOPS} = \frac{NUM\_OPS*x*y*z}{time}$

“ Please read our paper and see more experiments of

- 1D 2D stencils
- Double data type *”*

Evaluation

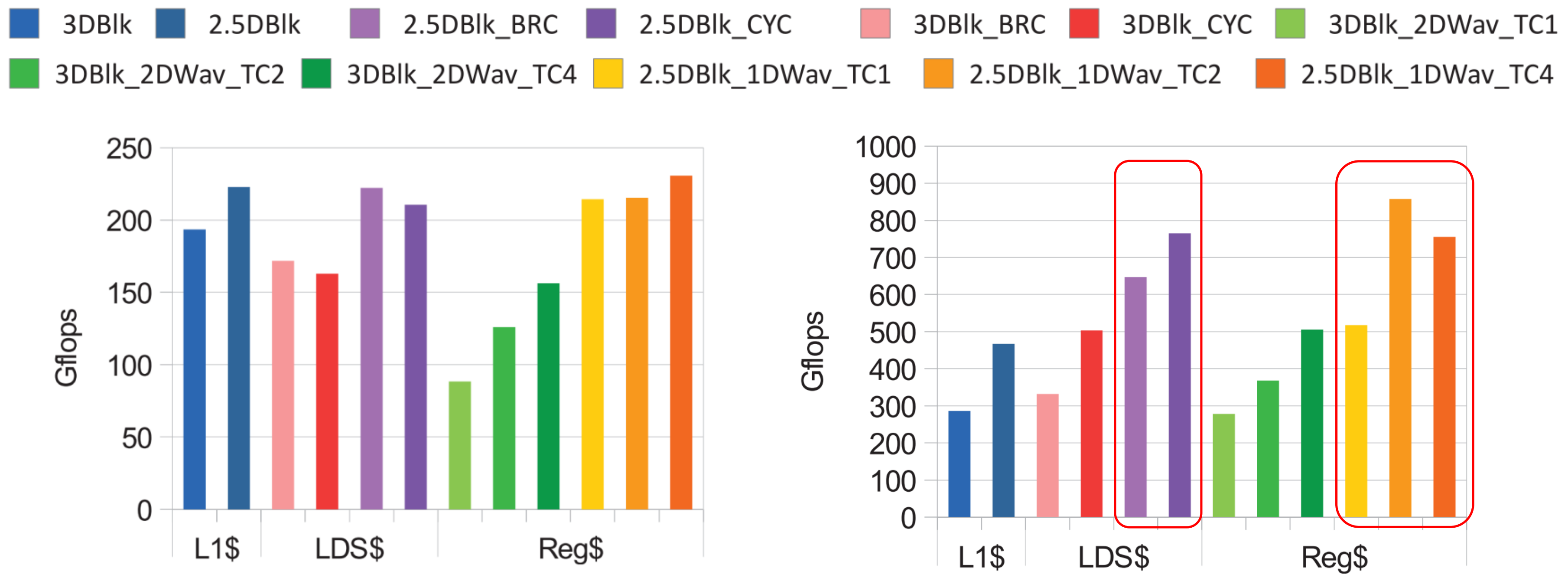
- Use different GPU-UNICACHE objects on AMD GPU



- 2.5D blocking is preferred in 3D stencils

Evaluation

- Use different GPU-UNICACHE objects on AMD GPU



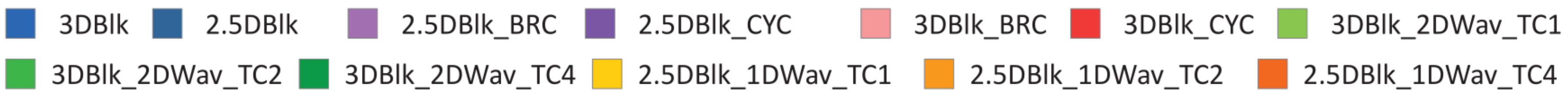
3D-7Point

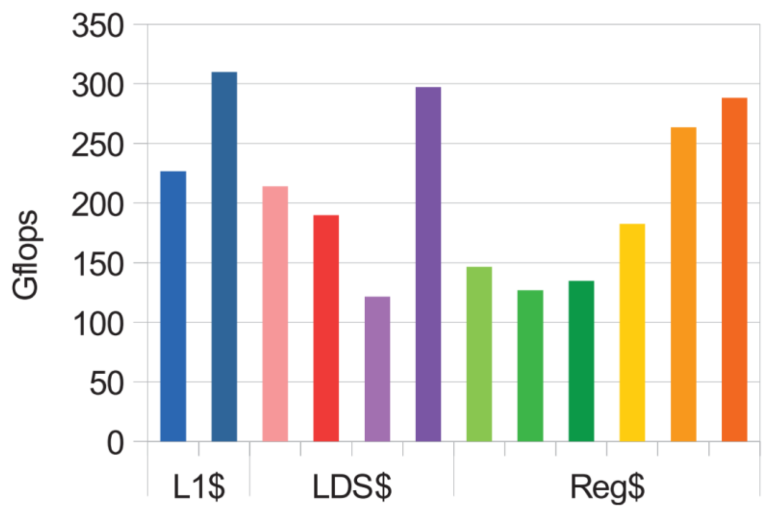
3D-27Point

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

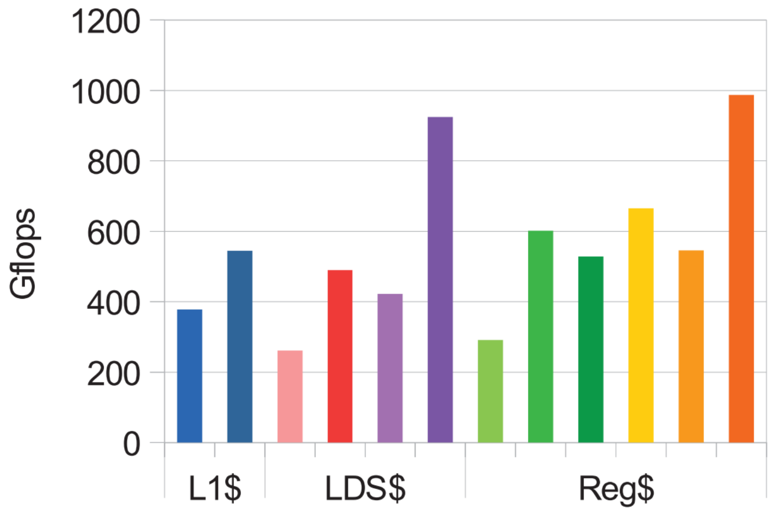
Evaluation

- Use different GPU-UniCache objects on NVIDIA GPU





3D-7Point

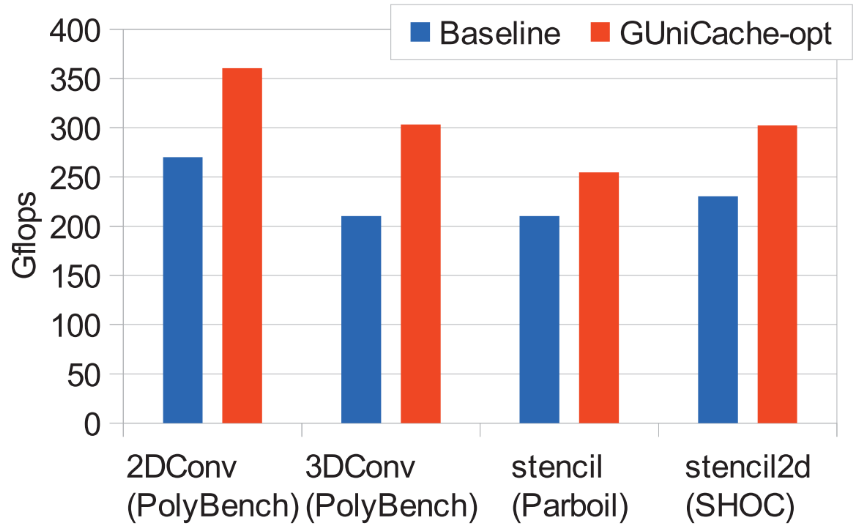


3D-27Point

- 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