High-Performance Computing for Stencil Computations Using a High-Level Domain-Specific Language

Justin Holewinski    Tom Henretty    Kevin Stock
Louis-Noël Pouchet    Atanas Rountev    P. Sadayappan

Department of Computer Science and Engineering
The Ohio State University

Presented at WOLFHPC 2011

May 31, 2011
Stencil Computations

- Operate on each point in a discrete $n$-dimensional space
- Use neighboring points in computation
- Often surrounded by time loop
- Have diverse boundary conditions

```c
for (i = 1; i < N-1; ++i) {
    for (j = 1; j < N-1; ++j) {
    }
}
```
Why do we need a domain-specific language?

- Easier for application developers and scientists
- Write stencil as point-function and grid instead of loop nest
- More opportunity for compiler optimization
- Restricted to a simple expression language
- Not restricted by C/C++/Fortran specification e.g. aliasing, memory life-cycle
- Control-flow is implicit instead of discovered at compile-time
- Iteration domain is easily obtained, enabling polyhedral transformations for tiling, parallelism, memory optimizations
- Computations on grids ease dependency analysis
Why do we need a domain-specific language?

- Easier for application developers and scientists
  - Write stencil as *point-function* and *grid* instead of loop nest
Why do we need a domain-specific language?

- Easier for application developers and scientists
  - Write stencil as *point-function* and *grid* instead of loop nest
- More opportunity for compiler optimization
  - Restricted to a simple expression language
  - Not restricted by C/C++/Fortran specification
    - e.g. aliasing, memory life-cycle
  - Control-flow is implicit instead of discovered at compile-time
  - Iteration domain is easily obtained, enabling polyhedral transformations for tiling, parallelism, memory optimizations
  - Computations on grids ease dependency analysis
Why do we need a domain-specific language?

- Easier for application developers and scientists
  - Write stencil as *point-function* and *grid* instead of loop nest
- More opportunity for compiler optimization
  - Restricted to a simple expression language
  - Not restricted by C/C++/Fortran specification
    e.g. aliasing, memory life-cycle
  - Control-flow is implicit instead of discovered at compile-time
  - Iteration domain is easily obtained, enabling polyhedral
    transformations for tiling, parallelism, memory optimizations
  - Computations on grids ease dependency analysis

Goal

*Use high-level abstractions to achieve write-once performance portability for stencil computations.*
Domain-Specific Language for Stencils

Stencil Compiler Workflow
Domain-Specific Language for Stencils

Define stencil operation as a *point-function* over a *grid* using

\([\text{time}]\text{grid}[i\text{-offset}][j\text{-offset}]\) notation:

```plaintext
1 pointfunction five_point_avg(p) {
2   float ONE_FIFTH;
3   ONE_FIFTH = 0.2;
4   [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1] + [0]p[0][0]
5       + [0]p[0][1] + [0]p[1][0]);
6 }
```
Define stencil operation as a *point-function* over a *grid* using 
\([\text{time}]\text{grid}[^{i\text{-offset}}][^{j\text{-offset}}]\) notation:

```
1 pointfunction five_point_avg(p) {
2   float ONE_FIFTH;
3   ONE_FIFTH = 0.2;
4   [1]p[0][0] = ONE_FIFTH*( [0]p[-1][0] + [0]p[0][-1] + [0]p[0][0] 
5       + [0]p[0][1] + [0]p[1][0]);
6 }
```

Define stencil range, functions, and convergence:

```
1 iterate 1000 {
2   stencil jacobi_2d {
3       [0][0:Nx-1] : [1]a[0][0] = [0]a[0][0];
4       [Ny-1][0:Nx-1] : [1]a[0][0] = [0]a[0][0];
5       [0:Ny-1][0] : [1]a[0][0] = [0]a[0][0];
6       [0:Ny-1][Nx-1] : [1]a[0][0] = [0]a[0][0];
7       [1:Ny-2][1:Nx-2] : five_point_avg(a);
8   }
9
10   reduction max_diff max {
11       [0:Ny-1][0:Nx-1] : [1]a[0][0] - [0]a[0][0];
12   }
13 } check (max_diff < .00001) every 4 iterations
```
Domain-Specific Language for Stencils

```
int Nx;
int Ny;
grid g [Ny][Nx];

float griddata a on g at 0,1;

pointfunction five_point_avg(p) {
  float ONE_FIFTH;
  ONE_FIFTH = 0.2;
  [1]p[0][0] = ONE_FIFTH*([0]p[-1][0] + [0]p[0][-1] + [0]p[0][0]
                      + [0]p[0][1] + [0]p[1][0]);
}

iterate 1000 {
  stencil jacobi_2d {
    [0][0:Nx-1] : [1]a[0][0] = [0]a[0][0];
    [Ny-1][0:Nx-1] : [1]a[0][0] = [0]a[0][0];
    [0:Ny-1][0] : [1]a[0][0] = [0]a[0][0];
    [0:Ny-1][Nx-1] : [1]a[0][0] = [0]a[0][0];
    [1:Ny-2][1:Nx-2] : five_point_avg(a);
  }

  reduction max_diff max {
    [0:Ny-1][0:Nx-1] : [1]a[0][0] - [0]a[0][0];
  }
}
check (max_diff < .00001) every 4 iterations
```

Complete stencil program
Optimizing CPU vs. GPU Performance

Floating-Point Throughput

- Need fine-grain and coarse-grain parallelism
Optimizing CPU vs. GPU Performance

Floating-Point Throughput

- Need fine-grain and coarse-grain parallelism
- On a CPU
  - Use vector processing units (SIMD)
  - Use threads to exploit multi-/many-cores

But this is not the whole story...
Optimizing CPU vs. GPU Performance

Floating-Point Throughput

▶ Need fine-grain and coarse-grain parallelism
▶ On a CPU
  ▶ Use vector processing units (SIMD)
  ▶ Use threads to exploit multi-/many-cores
▶ On a GPU
  ▶ Exploit SIMT parallelism across hundreds of cores
  ▶ Multiprocessors operate in lock-step ⇒ divergence = BAD

But this is not the whole story...
Optimizing CPU vs. GPU Performance

Floating-Point Throughput

- Need fine-grain and coarse-grain parallelism
- On a CPU
  - Use vector processing units (SIMD)
  - Use threads to exploit multi-/many-cores
- On a GPU
  - Exploit SIMT parallelism across hundreds of cores
  - Multiprocessors operate in lock-step $\Rightarrow$ divergence $=$ BAD

But this is not the whole story...
Memory Hierarchy

- Increasingly complex (multiple levels)
- Fast but small on-chip memory
- Slow but abundant off-chip memory
Memory Hierarchy

- Increasingly complex (multiple levels)
- Fast but small on-chip memory
- Slow but abundant off-chip memory
- On a CPU
  - Exploit hardware caches through data re-use
Memory Hierarchy

- Increasingly complex (multiple levels)
- Fast but small on-chip memory
- Slow but abundant off-chip memory
- On a CPU
  - Exploit hardware caches through data re-use
- On a GPU
  - Exploit per-multiprocessor shared/local memory
  - Maximize work per read/write operation

Need time tiling to efficiently utilize available main memory bandwidth
Optimizing Stencils on GPUs

Typical Approach

▶ Use spatial tiling to distribute work among thread blocks
▶ Use shared/local memory as program-controlled cache

Problems

▶ Global (off-chip) memory latency is high
▶ Limited data re-use within a thread block
▶ Cannot schedule enough threads to hide memory latency
▶ Traditional time tiling is not efficient due to branch divergence and a lack of memory access coalescing

Result

▶ Compute units are mostly idle waiting for memory operations to complete

A possible solution?

▶ Overlapped tiling

Holewinski et al.: Stencil Computations with a High-Level DSL
HPC Research Laboratory, The Ohio State University
Optimizing Stencils on GPUs

Typical Approach
- Use spatial tiling to distribute work among thread blocks
- Use shared/local memory as program-controlled cache

Problems
- Global (off-chip) memory latency is high
- Limited data re-use within a thread block
- Cannot schedule enough threads to hide memory latency
- Traditional time tiling is not efficient due to branch divergence and a lack of memory access coalescing
Optimizing Stencils on GPUs

Typical Approach
- Use spatial tiling to distribute work among thread blocks
- Use shared/local memory as program-controlled cache

Problems
- Global (off-chip) memory latency is high
- Limited data re-use within a thread block
- Cannot schedule enough threads to hide memory latency
- Traditional time tiling is not efficient due to branch divergence and a lack of memory access coalescing

Result
- Compute units are mostly idle waiting for memory operations to complete
Optimizing Stencils on GPUs

Typical Approach
- Use spatial tiling to distribute work among thread blocks
- Use shared/local memory as program-controlled cache

Problems
- Global (off-chip) memory latency is high
- Limited data re-use within a thread block
- Cannot schedule enough threads to hide memory latency
- Traditional time tiling is not efficient due to branch divergence and a lack of memory access coalescing

Result
- Compute units are mostly idle waiting for memory operations to complete

A possible solution?
- Overlapped tiling
Replace inter-tile communication with redundant computation

- Tile borders are redundantly computed by all neighboring tiles
- Trades extra FLOPs for a decrease in needed synchronization
- Enables time tiling without skewing (introduces divergence, load imbalance, and bank conflicts)
Overlapped Tiling

Replace inter-tile communication with redundant computation

- Tile borders are redundantly computed by all neighboring tiles
- Trades extra FLOPs for a decrease in needed synchronization
- Enables time tiling without skewing (introduces divergence, load imbalance, and bank conflicts)

Originally proposed by Krishnamoorthy et al. for parallelization

- We want fully-automatic code generation for arbitrary stencils
- Use OpenCL for performance-portable code generation, but tune parameters for different GPU architectures
Replace inter-tile communication with redundant computation

- Tile borders are redundantly computed by all neighboring tiles
- Trades extra FLOPs for a decrease in needed synchronization
- Enables time tiling without skewing (introduces divergence, load imbalance, and bank conflicts)

Originally proposed by Krishnamoorthy et al. for parallelization

- We want fully-automatic code generation for arbitrary stencils
- Use OpenCL for performance-portable code generation, but tune parameters for different GPU architectures

Let us look at an example for a $2 \times 2$ tile with a time tile size of 2...
Overlapped Tiling

Tile at time $t + 1$
Overlapped Tiling

Computed in time step t

Data needed at time t +1
Overlapped Tiling

Also computed by neighboring tiles

Computation at time t
Overlapped Tiling

Halo/Shadow data

Data needed at time t
Overlapped Tiling

GPU Implementation

- Schedule extra threads for redundant border cell computations
  - In general, need \((n + 2 \cdot r \cdot (t - 1)) \times (m + 2 \cdot r \cdot (t - 1))\) threads

Holewinski et al.: Stencil Computations with a High-Level DSL

HPC Research Laboratory, The Ohio State University
Overlapped Tiling

GPU Implementation

▶ Schedule extra threads for redundant border cell computations
  ▶ In general, need \((n + 2 \times r \times (t - 1)) \times (m + 2 \times r \times (t - 1))\) threads

▶ Use shared memory to store results across time
  ▶ Only need to access global memory in first and last time step of tile
Overlapped Tiling

GPU Implementation

- Schedule extra threads for redundant border cell computations
  - In general, need $(n + 2 \times r \times (t - 1)) \times (m + 2 \times r \times (t - 1))$ threads

- Use shared memory to store results across time
  - Only need to access global memory in first and last time step of tile

- Synchronize threads, not blocks, after each time step
  - Thread synchronization efficiently supported in hardware; block synchronization is not

Holewinski et al.: Stencil Computations with a High-Level DSL
HPC Research Laboratory, The Ohio State University
Overlapped Tiling

GPU Implementation

▶ Schedule extra threads for redundant border cell computations
  ▶ In general, need \((n + 2 \times r \times (t - 1)) \times (m + 2 \times r \times (t - 1))\) threads

▶ Use shared memory to store results across time
  ▶ Only need to access global memory in first and last time step of tile

▶ Synchronize threads, not blocks, after each time step
  ▶ Thread synchronization efficiently supported in hardware; block synchronization is not

▶ Use host to synchronize across time tiles

```c
for(t = 0; t < TIME_STEPS; t += TIME_TILE_SIZE) {
    invoke_kernel(input, output);
    swap(input, output);
    // Implicit barrier
}
```
What about block size?

Block size considerations

- Block size has large impact on performance
- Need enough threads to keep compute units busy...
What about block size?

Block size considerations
- Block size has large impact on performance
- Need enough threads to keep compute units busy...
- ... but it is also beneficial to use smaller blocks to increase the number of available registers per block
What about block size?

Block size considerations

- Block size has large impact on performance
- Need enough threads to keep compute units busy...
- ... but it is also beneficial to use smaller blocks to increase the number of available registers per block
- Problem size: $4096 \times 4096 \times 256$

![NVidia Tesla C2050 (GFlop/s)](image1)

![AMD Radeon HD 6970 (GFlop/s)](image2)
Arithmetic intensity matters too...

OpenCL on GPUs (4096x4096x256)

- **Base (9 FLOP/pt)**
- **Overlapped (9 FLOP/pt)**
- **Base (17 FLOP/pt)**
- **Overlapped (17 FLOP/pt)**
Performance

Jacobi 5-pt on Multi-Core with OpenMP

Jacobi 5-pt on GPU with OpenCL

- Fixed CPU tile sizes
- Fixed GPU block/tile sizes
Conclusion

- A DSL for stencils enables high productivity and performance
  - Higher-level for application developers
  - More information for compilers
  - Increased performance-portability

- Overlapped tiling enables high-performance stencils on GPUs
  - Trade redundant computation for less communication
  - Exploit high compute-per-memory-op ratio on GPUs
Performance Evaluation

Jacobi 5-pt on CPU and GPU (4096x4096x256)

GPU Block Size: $64 \times 8$ (512 of 1024 max)
Performance Evaluation

OpenMP on CPUs (4096x4096x256)

OpenCL on GPUs (4096x4096x256)

FP Through-put for Jacobi 9-pt
Performance Evaluation

Jacobi 5-pt on GPUs

Problem Size Evaluation for GPUs