Optimized Two-Level Parallelization for GPU Accelerators using the Polyhedral Model

CC 2017 - Austin, Texas
February 5th, 2017
Jun Shirako, Akihiro Hayashi, Vivek Sarkar
GPU Computing

- **Graphics Processing Units (GPUs)**
  - Significant performance and energy efficiency
  - Large burden on programmers due to low-level programming (e.g., CUDA and OpenCL)
    - Efficient parallelization for thousands of clustered cores
    - Explicit managements of data transfer and shared/local memories
    - Device-specific, low performance portability & productivity
- **High-level abstractions for GPU programming**
  - Users: programming in simple & platform-independent manner
  - Compilers: optimize/customize code for specific target systems
Polyhedral model

- Algebraic framework for affine program optimizations
  - Unified view that captures arbitrary loop structures
  - Generalized loop transformations as form of affine transform
  - Significant advancements over traditional AST-based transformations

Polyhedral compilation for GPUs (focus area for this work)

- End-to-end frameworks
  - C-to-CUDA [M. Baskaran, et al., CC 2010]
  - R-Stream [A. Leung, et al., GPGPU 2010]
  - PPCG [S. Verdoolaege, et al., TACO 2013]
- Input: sequential C. Output: optimized CUDA/OpenCL.
• **Two-level GPU parallelism**
  - Blocks: No/limited inter-block synchronization
  - Threads: Inter-thread barrier supported within a block
    - Coalesced memory accesses — i.e., contiguous threads to access contiguous elements — are critical to performance

• **Memory hierarchy management**
  - Explicit data transfers between global and shared (local) memories
Past work: PPCG Polyhedral Optimizer

- Coarse-grained parallelization policy [TACO 2013]
  - Compute **schedule — i.e., transformations** — based on PLuTo algorithm
  - Map the outermost parallelism in schedule to both blocks & threads
    - Fundamentally same parallelization for blocks & threads

```c
// Input (variant of Jacobi-2d)
for (t = 0; t < T; t++) {
    for (i = 1; i < N-1; i++)
        for (j = 1; j < N-1; j++)
    for (i = 1; i < N-1; i++)
        for (j = 1; j < N-1; j++)
            A[i][j] = B[i][j];
}
```

- **i-loop**
  - synchronization-free forall
- **j-loop**
  - cross-iteration dependence
  - accessing inner array dimension

```c
// Output of PPCG (CUDA kernel)
for (c1 = 0; c1 <= T-1; c1+=32) // t-tile
    for (c2 = 2 * c1; c2 <= ...; c2+=32) {
        if (...)
            for (c4 = ...; c4 <= ...; c4+=1) // t
                for (c5 = ...; c5 <= ...; c5+=1) {
                    if (N + 2 * c1 + 2 * c4 >= c2 + c5 + 2)
                        B[32*b0+t0][-2*c1+c2-2*c4+c5] = ...;
                    if (g7+c3 >= 2*g5+2*c2+2)
                        A[32*b0+t0][-2*g5+g7-2*c2+c3-1] = ...;
                }
    }
```

mapped to blocks & threads

sequentially executed; absence of memory coalescing
• Existing polyhedral approaches to GPUs
  • Compute *schedule — i.e., transformations* — based on PLuTo algorithm
  • Map the outermost parallelism in schedule to blocks & threads
    • Fundamentally same optimizations between block and thread
  • Block-level: synchronization-free parallelism is mandatory
  • Thread-level: can include barriers, important to coalesce memory accesses

• Our approach: two-level parallelization for GPUs
  • Compute *two schedules* with different optimization policies
    • Block-level: outermost synchronization-free parallelism
    • Thread-level: *parallelism with good coalescing + inter-thread synchronizations*
  • *Superposition* to integrate block-level and thread-level schedules
  • *DL memory cost model* to maximize coalesced memory access for threads
Outline

• Introduction
• Background
  • Overview of polyhedral model
  • Polyhedral transformations and parallelization
• Optimization framework
  • Overview of optimization flow
  • Superposition for GPU two-level parallelizations
  • GPU memory cost model for thread-level transformations
• Experimental results
• Conclusions
• Polyhedral model
  • Algebraic framework for affine program representations & transformations
    • Unified view that captures arbitrary loop structures
    • Generalize loop transformations as form of affine transform
  • Polyhedral representations (SCoP format)
    • Domain $D^Si$ : set of statement instances for statement $Si$
    • Access $A^Si$ : mapping an instance to array element(s) to be accessed
    • Schedule $\Theta^Si$ : mapping an instance to lexicographical time stamp
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        $C[i][j] = 0.0$;

for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < K; k++)

$D^{S1} = \{(i, j) : 0 \leq i \leq M-1, 0 \leq j \leq N-1\}$

$D^{S2} = \{(i, j, k) : 0 \leq i \leq M-1, 0 \leq j \leq N-1, 0 \leq k \leq K-1\}$

- Domain $D^{S_i}$: set of statement instances for statement $S_i$
Schedule (time mapping)

for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        \( S1: \ C[i][j] = 0.0; \)
    i:
        for (i = 0; i < M; i++)
    j:
        for (j = 0; j < N; j++)
            k: for (k = 0; k < K; k++)
        \( S2: \ C[i][j] = C[i][j] + A[i][k] \ast B[k][j]; \)

\( \Theta^{S1}(i, j) = (0, i, j) \)
\( \Theta^{S2}(i, j, k) = (1, i, j, k) \)

- **Schedule \( \Theta^{Si}(i) \):** mapping statement instance \( i \) to time stamp vector
  - To capture the sequential execution order of a program
  - Statement instances are executed in lexicographical order of schedules
Schedule (time mapping)

for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        S1:
            C[i][j] = 0.0;
        for (i = 0; i < M; i++)
            for (j = 0; j < N; j++)
                for (k = 0; k < K; k++)
                    S2:
                        C[i][j] = C[i][j] + A[i][k] * B[k][j];

θ^S1(i, j) = (0, i, j)
θ^S2(i, j, k) = (1, i, j, k)

for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        S1:
            C[i][j] = 0.0;
        for (k = 0; k < K; k++)
            S2:
                C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
}

codegen

• Schedule θ^Si(i): mapping statement instance i to time stamp vector
  • To capture the sequential execution order of a program
  • Statement instances are executed in lexicographical order of schedules
  • Transformation = find a new schedule under dependence constraints

θ^S1(i, j) = (0, i, j, 0)
θ^S2(i, j, k) = (0, i, j, 1, k)
Space-mapping

// high-level forall
forall (i = 0; i < M; i++)
forall (j = 0; j < N; j++)
S1: C[i][j] = 0.0;

// CUDA threads
i = threadIdx.y;
j = threadIdx.x;
S1: C[i][j] = 0.0;

Θ₁(i, j) = (i_y, j_x)

- Space-mapping Θᵣᵢ: mapping instance i to (logical) processor id
  - Represent parallelism
  - No sequential order among instances
  - Annotated with subscripts x, y, and z to represent GPU thread/block dimensions
Composition of Time- and Space-mapping

- Scattering function
  - In a multidimensional scattering function, some dimensions represent schedule (time-mapping) while others are space-mapping
  - Capture both sequential loop transformations and parallelization

\[
\Theta^{S_1} = (t, 0, i_y, j_x) \\
\Theta^{S_2} = (t, 1, i_y, j_x)
\]

* Space-mapping dimension is annotated with subscripts
Outline

• Introduction
• Background
  • Overview of polyhedral model
  • Polyhedral transformations and parallelization
• Optimization framework
  • Overview of optimization flow
  • Superposition for GPU two-level parallelizations
  • GPU memory cost model for thread-level transformations
• Experimental results
• Conclusions
Overall Flow

- Transformations and parallelization
  - Thread-level transformations
    - Extended memory cost model (DL model) to GPU memory warps
    - Detect loop parallelism with good coalesced memory access; map to the innermost thread dimension
  - Block-level transformations (independent of thread-level transformations)
    - Detect & map sync-free parallelism to block dimensions
  - Superposed into final scattering function

- Shared memory and register optimizations
  1. Individual tiles are identified after superposition
  2. Array elements to be used/modified within each tile are computed
  3. Insert data transfers to/from shared memory or registers
Superposition

• Two scattering functions per statement
  • Block-level scattering function, $\Theta_{\text{Sout}}(i)$
    • Many-to-one function to assign multiple instances $i$ to same value
      • Must be fully permutable
  • Thread-level scattering function, $\Theta_{S}(i)$
    • One-to-one function to assign each instance $i$ to a unique value

• Superposition as loop tiling
  • Block-level: specify inter-tile schedule (individual tiles)
  • Thread-level: specify intra-tile schedule (iterations within a tile)
Superposition

Schedule for original code:
\[ \Theta^{S_1} = (t, 0, i, j) \]
\[ \Theta^{S_2} = (t, 1, i, j) \]

Block scattering function:
\[ \Theta^{S_1\text{out}} = (i_x) \]
\[ \Theta^{S_2\text{out}} = (i_x) \]

Thread scattering function:
\[ \Theta^{S_1} = (t, 0, i_y, j_x) \]
\[ \Theta^{S_2} = (t, 1, i_y, j_x) \]

Superposed scattering function:
\[ \Theta^{S_1} = \left\lfloor \frac{i_x}{32} \right\rfloor, t, 0, i_y, j_x \]
\[ \Theta^{S_2} = \left\lfloor \frac{i_x}{32} \right\rfloor, t, 1, i_y, j_x \]

* Tile size 32 is used
Analytical Model for Coalescing Memory Access

- DL model for CPU memory cost analysis
  - Originally proposed for cache (and TLB)
  - Assumption: loop tiling is applied
    - All data per tile fits within target cache
  - DL = estimation of # distinct cache lines per tile
    - Function of tile sizes, $T_1, T_2, \ldots, T_d$
    - $DL(T_1, T_2, \ldots, T_d) \leq$ total cache miss count per tile
- Extensions to GPU memory warp
  - Additional assumption: shared memory transfer
    - per-tile data is optimally prefetched & kept on shared/cache memory
  - Extended DL = estimation of # memory transactions per tile
    - $DL(T_1, T_2, \ldots, T_d) \leq$ total memory transaction count per tile
Analytical Model for Coalescing Memory Access

\[
\text{DL}(T_1, T_2, T_3) = \text{DL}_A(T_1, T_2, T_3) + \text{DL}_B(T_1, T_2, T_3) = T_1 \times \left\lceil \frac{T_2}{L} \right\rceil + T_3 \times \left\lceil \frac{T_1}{L} \right\rceil
\]

\[
\text{mem\_cost}(T_1, T_2, \ldots, T_d) = \frac{\text{Cost}_{\text{trans}} \times \text{DL}(T_1, T_2, \ldots, T_d)}{T_1 \times T_2 \times \ldots \times T_d}
\]

L: warp size (e.g., 32 for NVIDIA GPUs), \( \text{Cost}_{\text{trans}} \): cost of single memory transaction

- Extended DL = estimation of # optimal memory transactions per-tile
  - Per-tile data is optimally prefetched & kept on shared/cache memory
  - Memory cost = total memory transaction count (normalized as per-iteration)
Profitability Analysis via DL Memory Cost

• Loop with best memory coalescing
  • Partial derivative of memory cost w.r.t. $T_k$:
    $$\frac{\partial \text{mem\_cost}(T_1, T_2, ..., T_d)}{\partial T_k}$$
  • Reduction rate of memory cost when increasing $T_k$
  • Parallel loop with most negative value
    $\rightarrow$ most profitable loop for memory cost minimization
    $\rightarrow$ mapped to innermost thread dimension

• Profitability of loop fusion
  • Comparing $\text{mem\_cost}(T_1, T_2, ..., T_d)$ before and after fusion
    • Memory cost decreased $\rightarrow$ fusion is profitable
  • Other criteria, e.g., loss of parallelism, are also considered
Experimental Setting

• **Platforms**
  - Intel Xeon X5660 + NVIDIA Tesla M2050 GPU (Fermi)
    - 13SM x 32-core, total 448 CUDA Cores
  - IBM POWER8 + NVIDIA Tesla K80 GPU (Kepler)
    - 14SMX x 192-core, total 2496 CUDA Cores

• **Benchmarks**
  - PolyBench-C 3.2
  - SPEC Accel : 314.omriq and 357.sp (two kernels from x_solve)

• **Experimental variants**
  - Sequential : gcc -O3 on CPU
  - PPCG : Polyhedral Parallel Code Generator from INRIA
  - PolyAST+GPU : Two-level parallelization for GPUs (proposed)
• block-level: PolyAST+GPU has same schedule as PPCG
• thread-level: different schedules due to superposition and coalescing policy
• PPCG has more efficient code generation method (e.g., # threads can be ≤ block size)
• Geometric mean speedup: 44.8× by PPCG and 85.9× by PolyAST+GPU
• Relative improvement of our work over PPCG ~ 1.8x
block-level: PolyAST+GPU has same schedule as PPCG
thread-level: different schedules due to superposition and coalescing policy
PPCG has more efficient code generation method (e.g., # threads can be ≤ block size)
Geometric mean speedup: 45.6× by PPCG and 95.5× by PolyAST+GPU
Relative improvement of our work over PPCG ~ 2.1x
Conclusions

• **Graphics Processing Units (GPUs)**
  - Massively parallel architecture consisting of thousands of cores
  - Large burdens upon programmers, comparing with SMP programming
  - Automatic C-to-CUDA transformations for productive GPU computing

• **Existing polyhedral approaches to GPUs**
  - Focus on sync-free parallelism; less attention to generating threads with barriers
  - Use same schedule for both blocks and threads

• **Two-level parallelizations for GPUs**
  - Allows block-level and thread-level schedules with different optimization policies
    - Superposition to integrate block-level and thread-level schedules
    - An analytical memory cost model for GPU memory warp analysis
  - $1.8 \times$ and $2.1 \times$ geometric mean improvements on NVIDIA Fermi and Kepler over PPCG