Program Optimization Study on a 128-Core GPU

Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, and Wen-mei W. Hwu

Yu, Xuan

Dept of Computer & Information Sciences

University of Delaware

CISC 879 : Software Support for Multicore Architectures

General Idea

- Good news
  - Improving programmability and generality on GPU
  - Possibility to perform a wide variety of parallelization optimizations
- Problem
  - How do you choose and control optimizations on GPU properly
    - Possible combination of optimization is very large and makes the optimization space tedious to explore
    - Limited local resources and global memory bandwidth makes performance sensitive to even small changes in code, unpredictable

General Idea

- Presented a study that examines a broad space of optimizations performed on several applications
- Found configurations up to 74% faster than previously thought optimal.
- Explained why this is happening on GPU, discuss some principles and techniques for finding near-optimal configurations

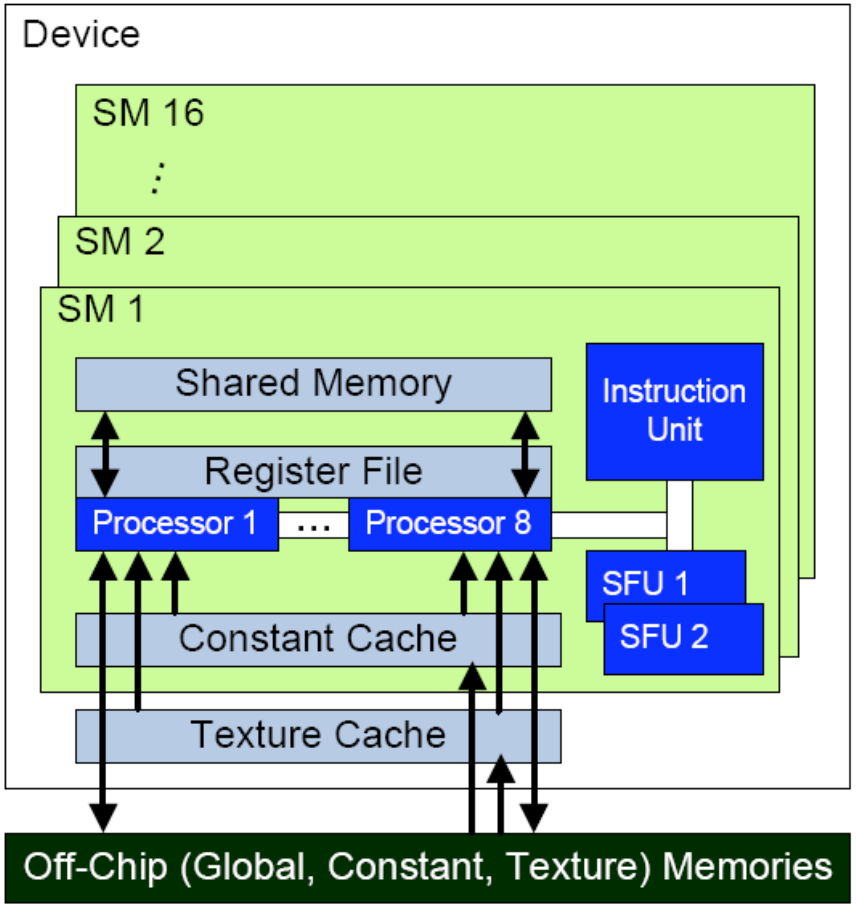
Organization

- Architecture Overview (CUDA)
  - Introduction of execution hardware and threading model
  - Compute Unified Device Architecture (CUDA)
- optimization space search
  - Discussion of the space search process and the classifications and characteristics of the program optimizations
- Experiments
  - Discuss result of the search for several applications
  - Matrix Multiplication
  - Magnetic resonance Imaging
  - Sums of Absolute Difference
- Conclusion

Architecture

- General Programming and compilation process
  - ✓ GPU is treaded as a coprocessor that executes data-parallel kernel functions
  - ✓ The user supplies a both host (CPU) and kernel (GPU) code
  - ✓ Codes are separated and compiled by NVIDIA's compiler.
  - ✓ Host code transfers data to GPU's and initialed the kernel code via API calls

Architecture



16 *streaming multiprocessors (SMs)*

Each SM containing eight *streaming processors (SPs), or cores*

Each core executes a single thread's instruction in SIMD

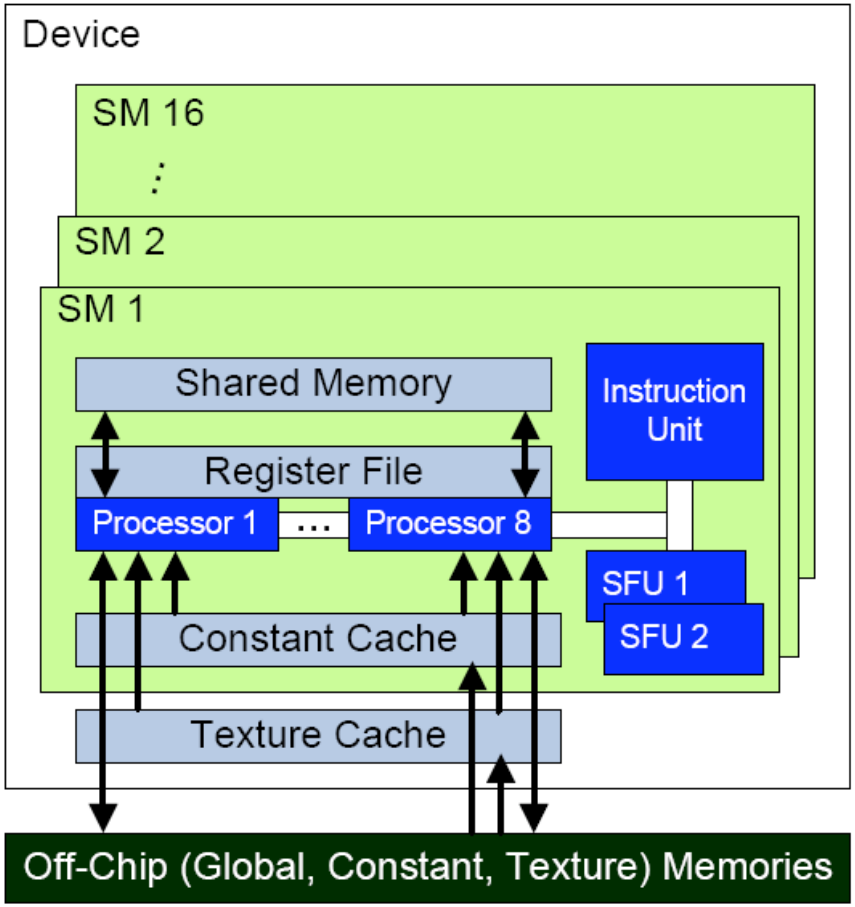
multiply-add arithmetic unit

two special functional units (SFUs)

reciprocal square root, sine, and cosine

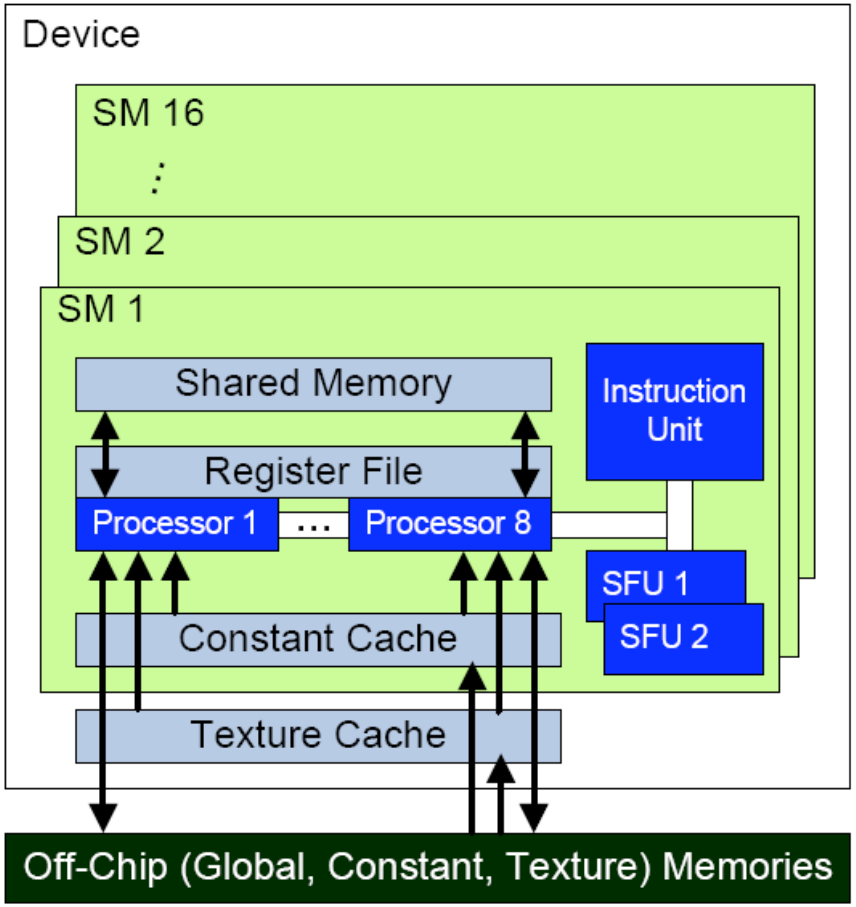
fully pipelined,

Architecture



Memory	Location	Size	Latency	Read-Only
Global	off-chip	768MB total	200-300 cycles	no
Shared	on-chip	16KB per SM	≃register latency	no
Constant	on-chip cache	64KB total	≃register latency	yes
Texture	on-chip cache	up to global	>100 cycles	yes
Local	off-chip	up to global	same as global	no

Architecture



Three Level Hierarchy:
•Grid
•Block
•Thread

Each kernel creates a single *grid*

A grid consists of many thread blocks.(512, on single SM)

Threads in a block are organized into warps of 32 threads. Each warp executes in SIMD fashion, issuing in four cycles on the eight SPs of an SM.

When one wrap stall, SM switch to another warp

Architectural Interactions

- Hardware constraints

Resource or Configuration Parameter	Limit
Threads per SM	768 threads
Thread Blocks per SM	8 blocks
32-bit Registers per SM	8,192 registers
Shared Memory per SM	16,384 bytes
Threads per Thread Block	512 threads

These constrains interacts with each other making accurately predicting the effects of one or more compiler optimizations of CUDA difficult.

Architectural Interactions

Consider an application:
•Uses 256 threads per block
•10 registers per thread
•4KB of shared memory per thread block.
Can schedule 3 thread blocks and 768 threads on each SM.

An optimization:
Increases each thread's register usage from 10 to 11 (an increase of only 10%) will decrease the number of blocks per SM from 3 to 2. This decreases the number of threads on an SM by 33%.
Why? 768 * 11 = 8448 > 9192

Resource or Configuration Parameter	Limit
Threads per SM	768 threads
Thread Blocks per SM	8 blocks
32-bit Registers per SM	8,192 registers
Shared Memory per SM	16,384 bytes
Threads per Thread Block	512 threads

Architectural Interactions

By contrast, an optimization that increases each thread block's shared memory usage by 1KB (an increase of 25%) does not decrease the number of blocks per SM. Clearly, the optimization space is inherently non-linear.

Resource or Configuration Parameter	Limit
Threads per SM	768 threads
Thread Blocks per SM	8 blocks
32-bit Registers per SM	8,192 registers
Shared Memory per SM	16,384 bytes
Threads per Thread Block	512 threads

Optimization space search

- Architecture Overview (CUDA)
  - Introduction of execution hardware and threading model
  - Compute Unified Device Architecture (CUDA)
- Optimization space search
  - Discussion of the space search process and the classifications and characteristics of the program optimizations
- Experiments
  - Discuss result of the search for several applications
  - Matrix Multiplication
  - Magnetic resonance Imaging
  - Sums of Absolute Difference
- Conclusion

Optimization space search

Basic strategy for good performance:

Reduce dynamic instruction count while maintaining high SP occupancy.

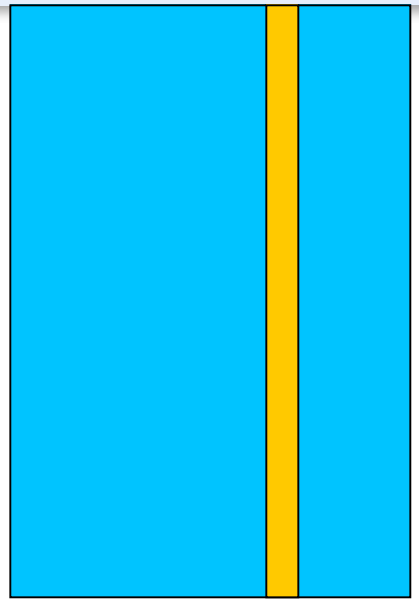
Four categories of machine-level behavior to optimizae

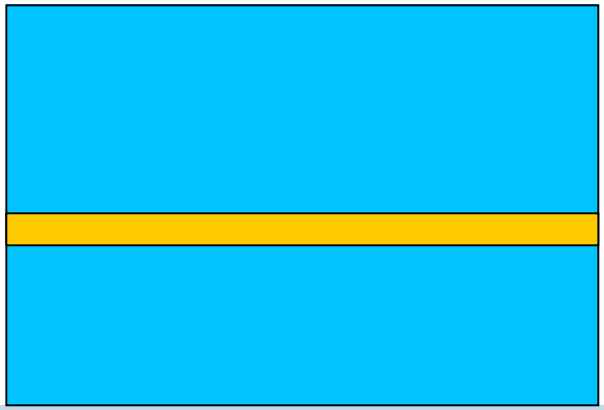
- Thread-level work redistribution
- Instruction count reduction
- Intra-thread parallelism
- Resource balancing

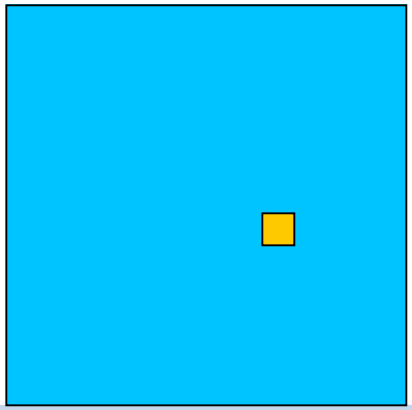
Optimization space search

Example of matrix multiplication

The kernel is tiled so that each thread block computes a square 16-by-16 tile of the output matrix







Optimization space search

Example of matrix multiplication

```
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
```

tx and ty are each thread's coordinates in the thread block;

indexA , indexB , and indexC are positions in the matrices

Threads in a block cooperatively load parts of the input matrices into shared memory, amortizing the cost of global load latency

Using larger tiles enhances the benefit of data sharing, but reduces scheduling flexibility since a greater fraction of the threads on an SM must wait at barrier synchronizations.

Optimization space search

Four categories of machine-level behavior to optimize

- Thread-level work redistribution
- Instruction count reduction
- Intra-thread parallelism
- Resource balancing

Each thread compute two matrix elements instead of one, presents opportunities for eliminating redundant instructions previously distributed across threads

```
Ctemp = Dtemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][32];

  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  Bs[ty][tx+16] = B[indexB+16];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
      Dtemp += As[ty][i]
        * Bs[i][tx + 16];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
C[indexC+16] = Dtemp;
```

Optimization space search

Four categories of machine-level behavior to optimize

- Thread-level work redistribution
- Instruction count reduction
- Intra-thread parallelism
- Resource balancing

Traditional compiler optimizations such as common sub expression elimination, loop-invariant code removal, and loop unrolling.

```
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  Ctemp +=
     As[ty][0] * Bs[0][tx];
  ...
  Ctemp +=
     As[ty][15] * Bs[15][tx];

  __syncthreads();
}
C[indexC] = Ctemp;
```

Optimization space search

Four categories of machine-level behavior to optimize

- Thread-level work redistribution
- Instruction count reduction
- Intra-thread parallelism
- Resource balancing

A developer can unroll loops to facilitate code scheduling in the compiler or explicitly insert pre-fetching code.

```
a = A[indexA];
b = B[indexB];
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  As[ty][tx] = a;
  Bs[ty][tx] = b;
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  a = A[indexA];
  b = B[indexB];
  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
```

(d) Prefetching

Optimization space search

Four categories of machine-level behavior to optimize

- Thread-level work redistribution
- Instruction count reduction
- Intra-thread parallelism
- Resource balancing

Trade certain resource usages, some of which may be counterintuitive, to produce a better performing application.

An example of this is using shared memory to buffer data for reuse, regardless of whether it is shared with other threads.

Another example is proactive register spilling by the programmer. By reducing register usage, often a critical resource, more thread blocks can be assigned to each SM.

Experiments

- Architecture Overview (CUDA)
  - Introduction of execution hardware and threading model
  - Compute Unified Device Architecture (CUDA)
- optimization space search
  - Discussion of the space search process and the classifications and characteristics of the program optimizations
- Experiments
  - Discuss result of the search for several applications
  - Matrix Multiplication
  - Magnetic resonance Imaging
  - Sums of Absolute Difference
- Conclusion

Experiments

Comparison:

GPU experiments:

AMD Opteron 248 2.2GHz with 1GB main memory.

CPU versions:

Intel Core2 Extreme Quad running at 2.66 GHz with 4GB main memory.

Application	Description	Max Speedup over CPU
Matrix Multiplication	Multiplication of two dense 4k x 4k matrices. The CPU version uses version 9.0 of the Intel C++ Compiler and version 8.0 of the Intel Math Kernel Library.	6.98X
SAD	Computation of sums of absolute differences. SADs are computed between 4x4 pixel blocks in two QCIF-size images over a 32 pixel square search area.	19.6X
MRI-Q	Computation of a matrix Q, representing the scanner configuration, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	351X
MRI-FHD	Computation of an image-specific matrix $F^H d$, used in a 3D magnetic resonance image reconstruction algorithm in non-Cartesian space.	220X

Experiments

We varied tiling sizes, tiling dimensions, pre-fetching, and unroll factors,

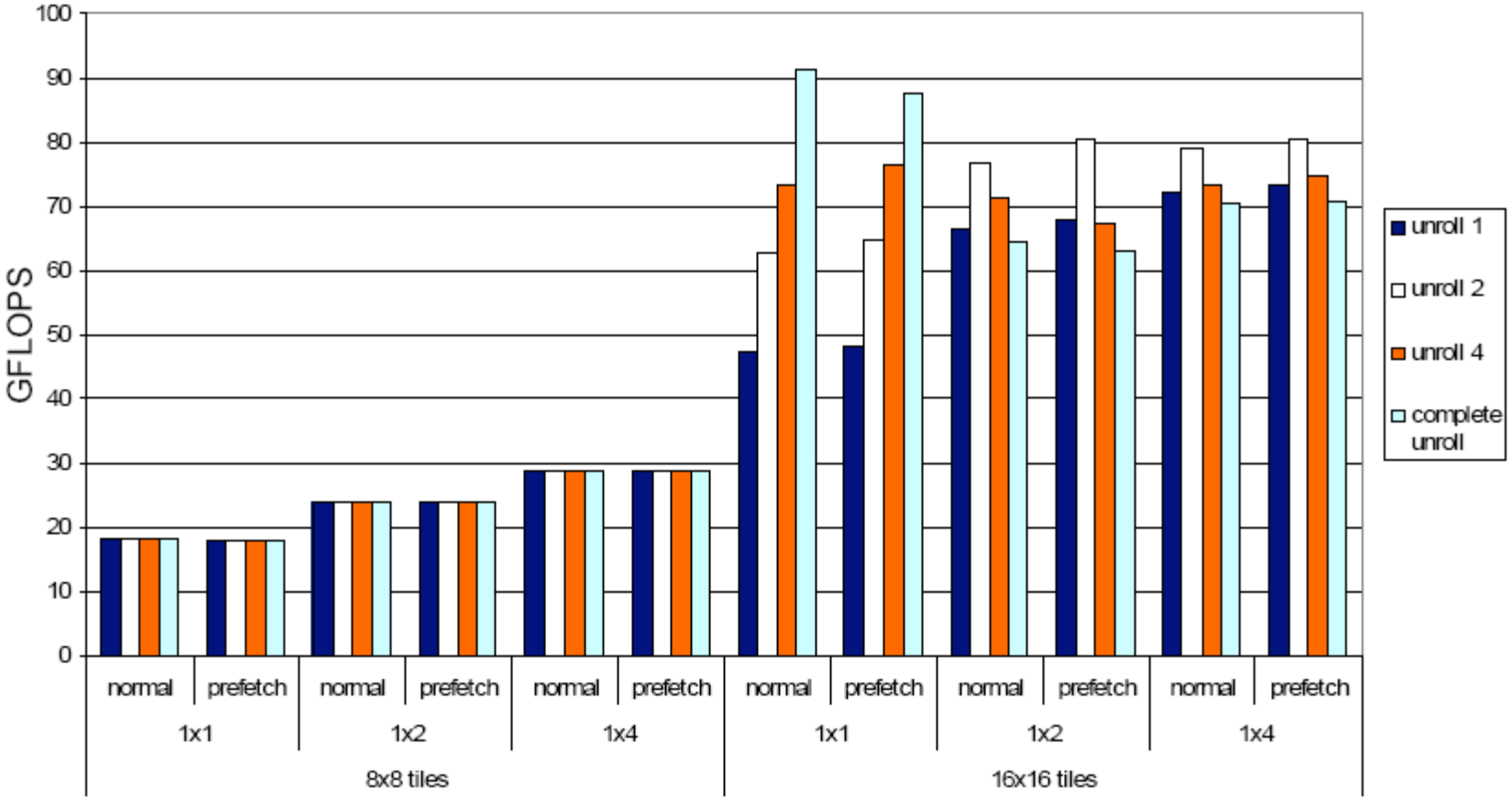


Fig. 3. Matrix Multiplication Results

Experiments

The general trend: Larger tiles sizes and more work per thread gives higher performance

- Initial thought optimal
  - 1x1 tiling, 16x16 tiles, complete unrolling, pre-fetching
  - 87.5 GFLOPS.
- Actual peak performing:
  - 1x1 tiling, 16x16 tiles, complete unrolling, no pre-fetching
  - 91.3 GFLOPS, an improvement of 4.2%.

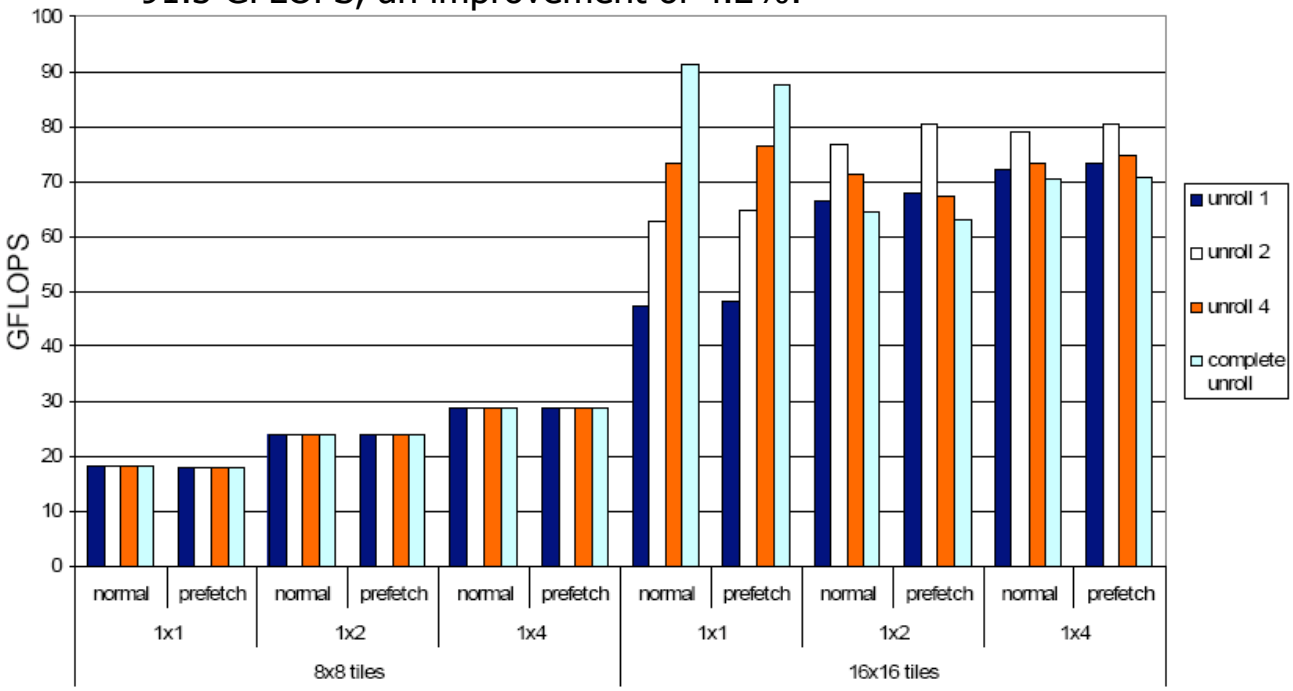


Fig. 3. Matrix Multiplication Results

Experiments

Increasing the tiling dimensions:

- Stable performance
- Slight advantage in average
- does not result in peak performance.

Reason: negative effects of unrolling by more than a factor of two for the higher tiling dimensions.

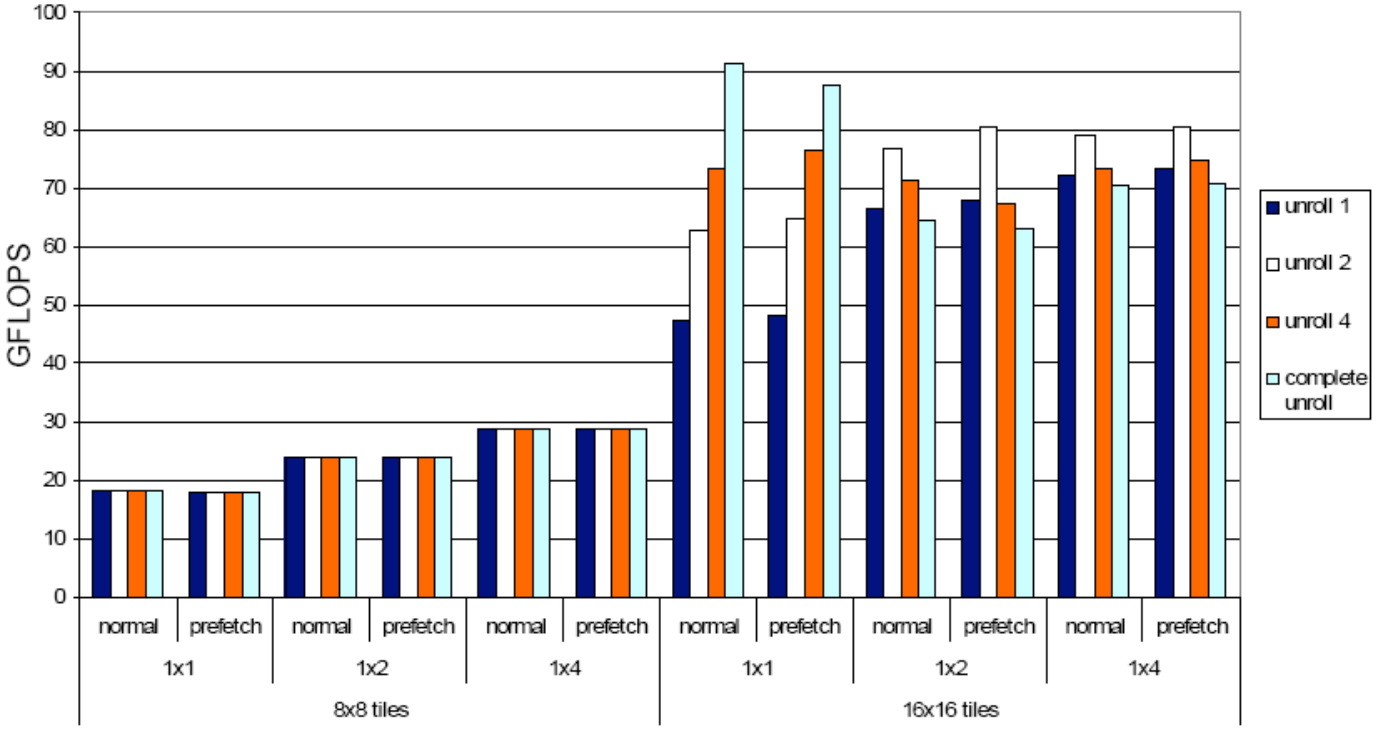


Fig. 3. Matrix Multiplication Results

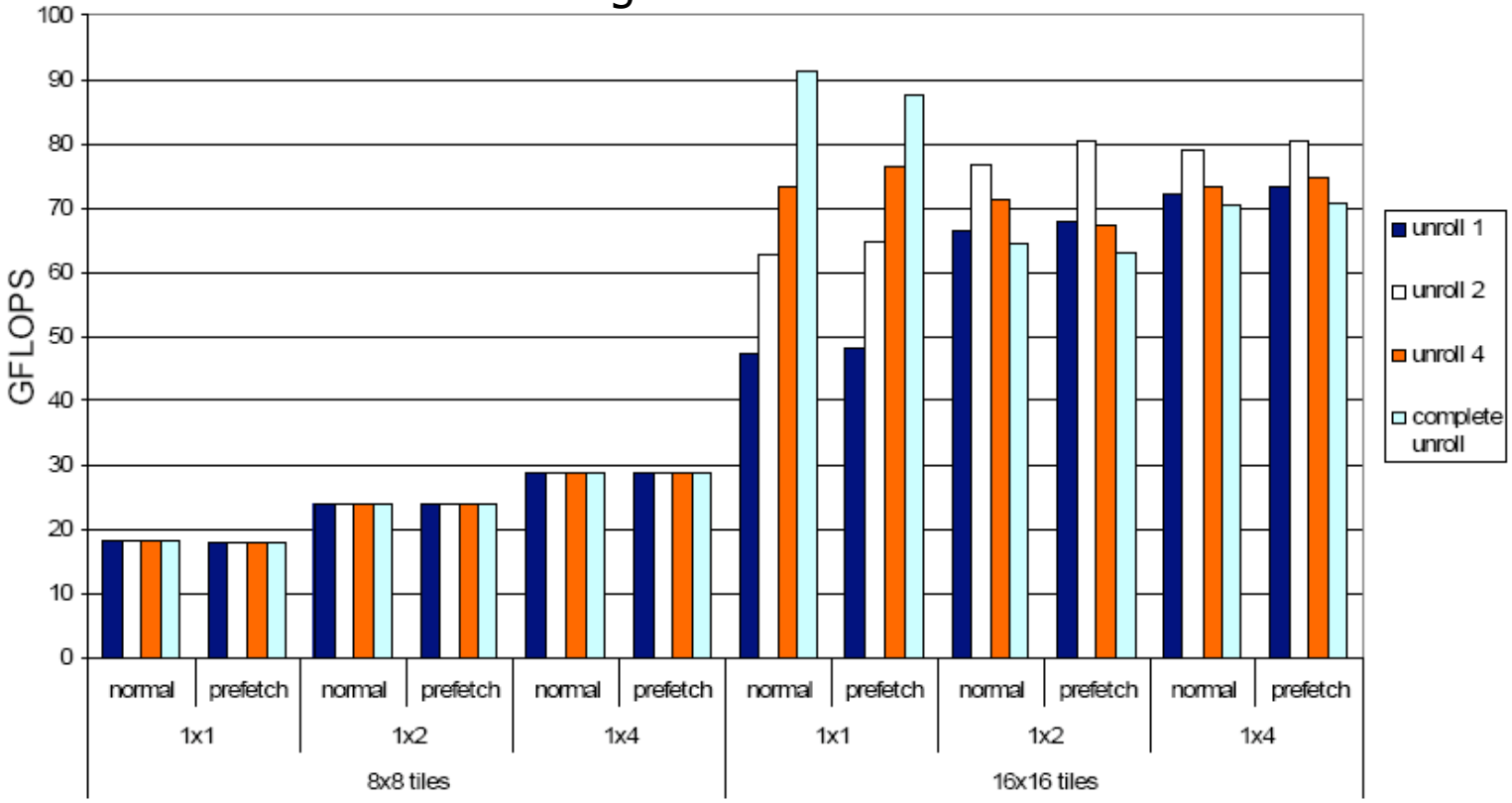
Experiments

In summary:

✓larger thread blocks are good due to data sharing.

✓Complete unrolling often good due to reducing the branches calculations.

❖However, the runtime's scheduling may increase register pressure such that the number of thread blocks assigned to each SM is reduced.



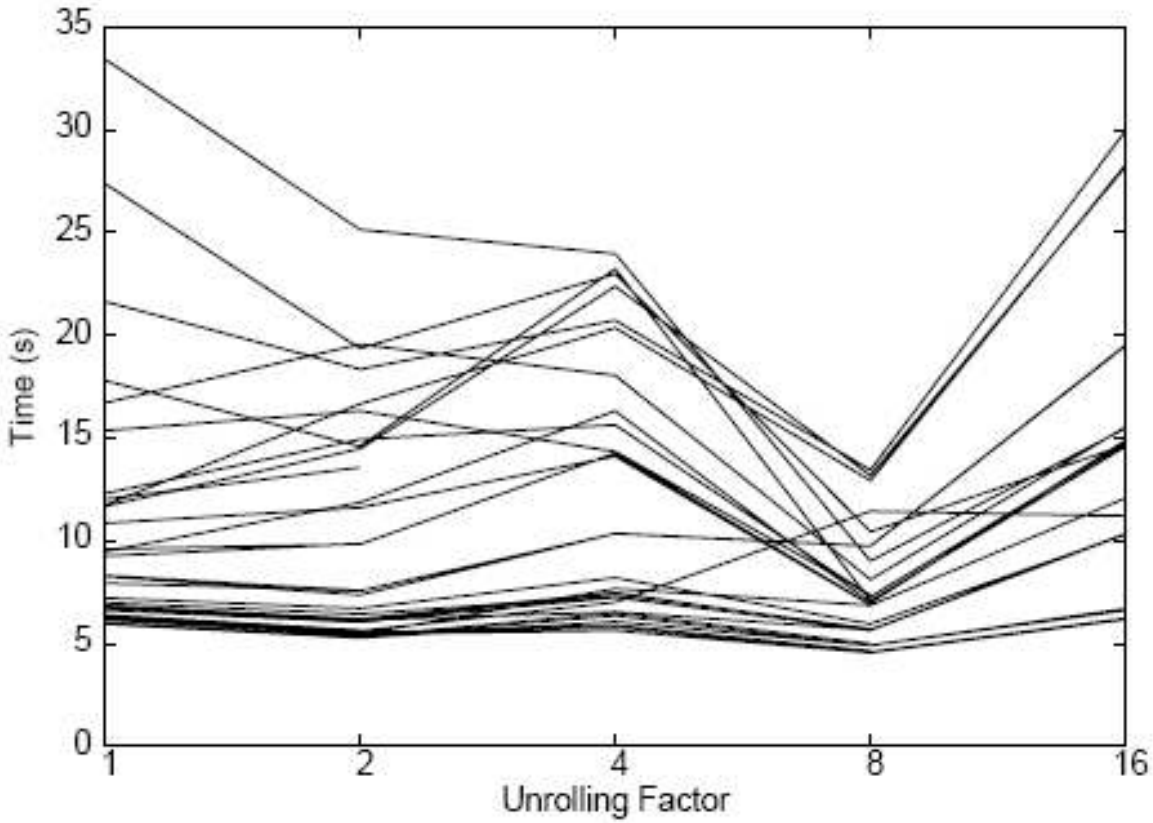
Experiments

Another application:
Magnetic resonance imaging (MRI) reconstruction

Reconstruct high-quality images from non-Cartesian trajectories. The computation required to perform these reconstructions is substantial.

Parameters sensitive to performance:

- loop unrolling factor,
- The number of threads per block (tpb),
- The number of scan points processed by each grid



✓Shorter execution time for an unrolling factor of 8

✓4 is often worse than either 2 or 8

Reason:

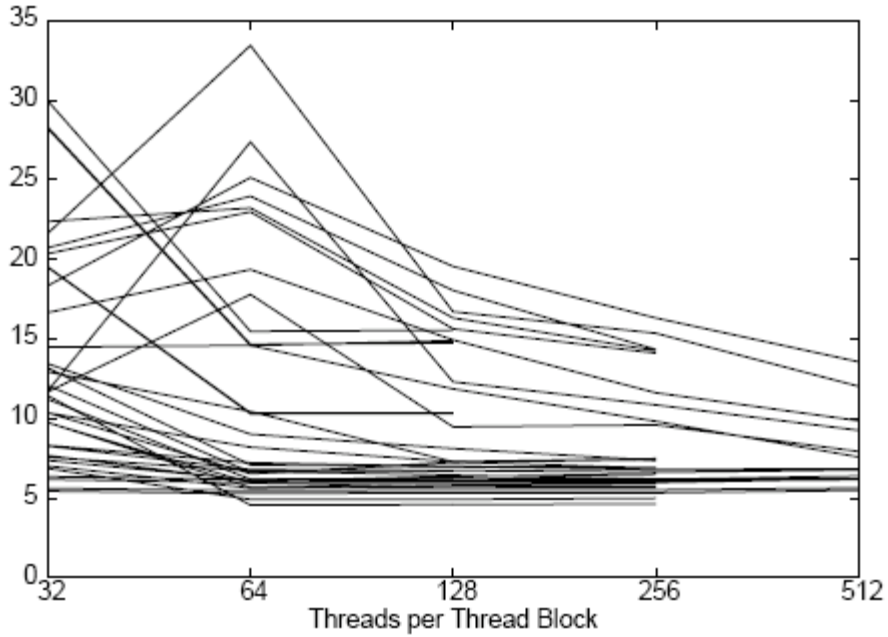
•12 registers when unrolled, 2 thread blocks per SM and 6.17s.

•12 registers Unrolling factor is 2, 5.52s.

•24 registers unrolling factor is 4, only admit on block per SM 5.89s

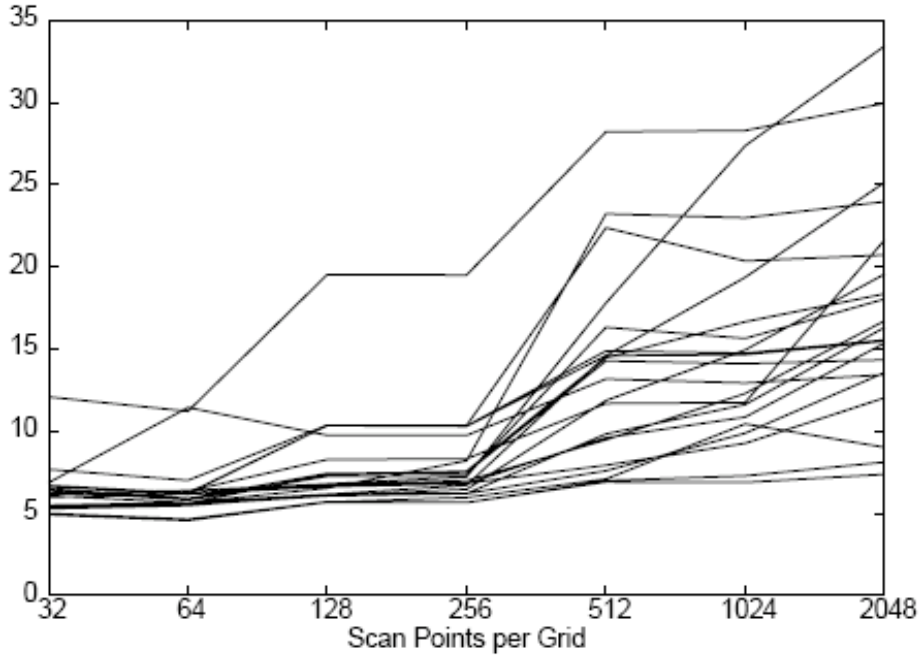
•30 registers unrolling factor is 8, 1 block per SM, 4.64s

Experiments



(b) Varying threads per thread block (tbp). Each line fixes unrolling factor and points per grid.

Experiments



(c) Varying scan points per grid (ppg). Each line fixes unrolling factor and threads per block.

there is a smaller chance of conflicts when fewer thread blocks run on an SM.

Experiments

To summarize MRI

Performance relatively insensitive to block size

An unrolling factor of 8 provided the highest performing

Conclusions

Gradual changes in optimization parameters can have wildly varying effects on an application.

Local resources used by a thread increases to points where fewer thread blocks can be assigned to each SM will reduce overall performance.

They believe that scheduling should be better controlled, possibly by the compiler rather than the runtime.