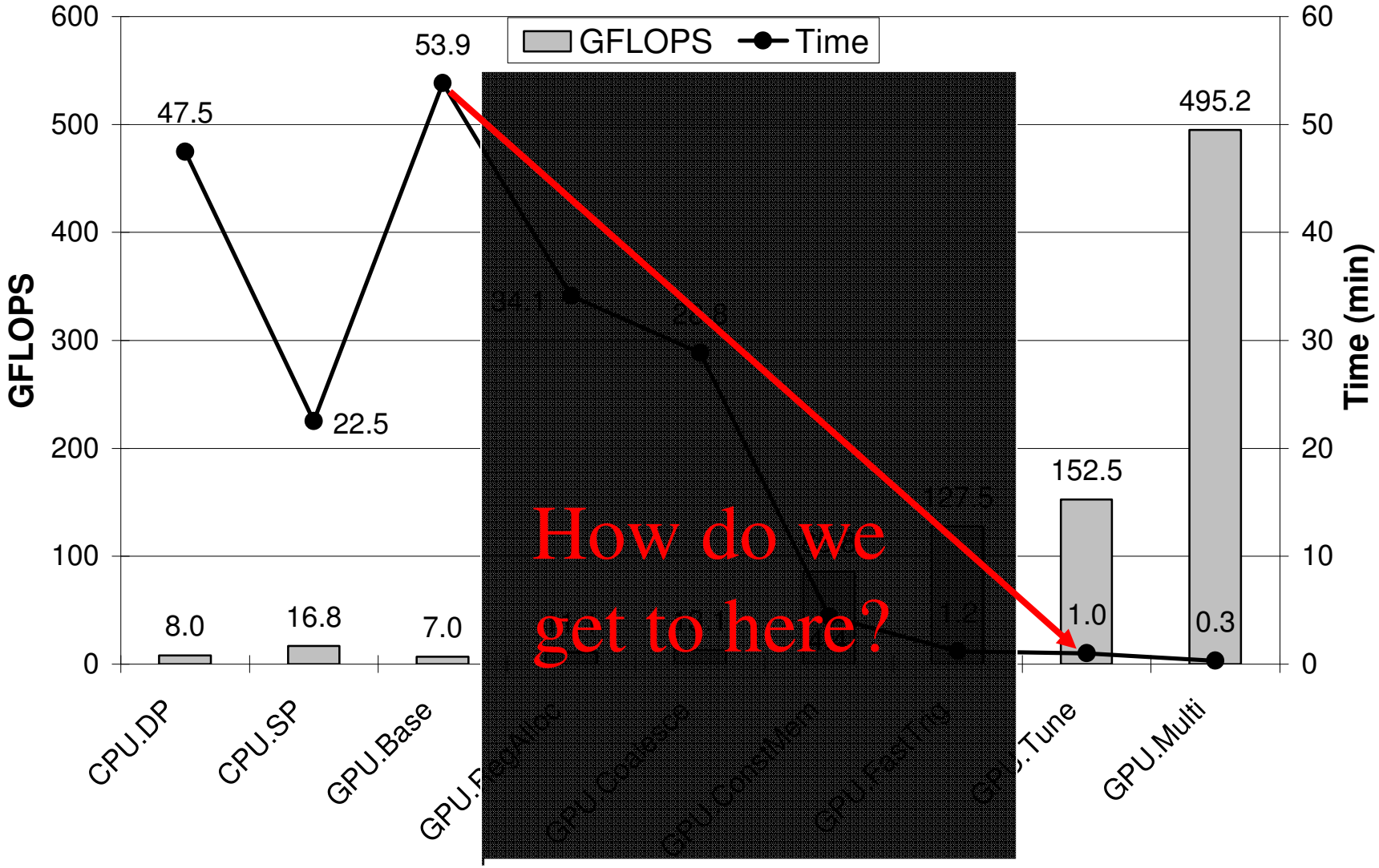
Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA

Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi,
Sam S. Stone, David B. Kirk, and Wen-mei H. Hwu

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign
and NVIDIA Corporation

MRI-F^{H}D Performance



Principles

- Leverage zero-overhead thread scheduling
- Inter-thread communication possible locally, not globally
- Optimize use of on-chip memory
- Group threads to avoid SIMD penalties and memory port/bank conflicts

- Further optimization involves tradeoffs between resources

Managing Memory Latency

- Global memory latency is 200+ cycles
- 8 instructions/cycle
- Need 1600 instructions to avoid stalling

- Decompose work into a fine granularity for TLP
- ILP and MLP within each thread have a multiplicative effect

```
Ctemp = 0;

for (i = 0; i < widthA;
     i++)
  {
    Ctemp += A[indexA]
      * B[indexB];
    indexA++;
    indexB += widthB;
  }

C[indexC] = Ctemp;
```

Matrix multiplication

Each thread – 1 result element

1024x1024 matrix:

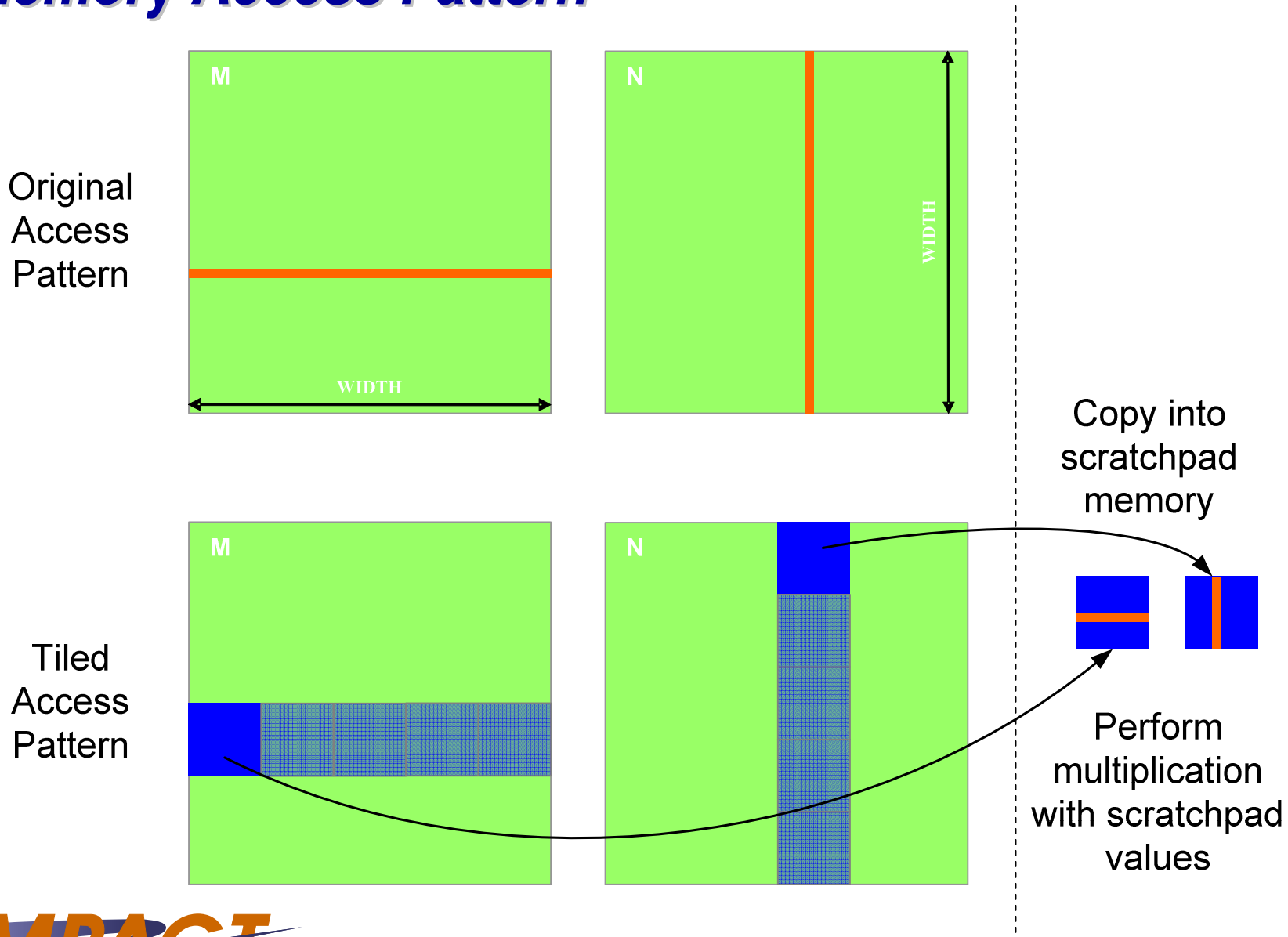
1M threads

Global Bandwidth Saturation

- 2 global loads for every 6 instructions
- Requires more than 2X the available bandwidth
- Inter-thread data reuse: local scratchpad memory can reduce global bandwidth usage

```
Ctemp = 0;

for (i = 0; i < widthA;
     i++)
  {
    Ctemp += A[indexA]
      * B[indexB];
    indexA++;
    indexB += widthB;
  }

C[indexC] = Ctemp;
```

Memory Access Pattern



Reducing Memory Bandwidth Usage

```
Ctemp = 0;

for (i = 0; i < widthA;
     i++)
{
  Ctemp += A[indexA]
    * B[indexB];
  indexA++;
  indexB += widthB;
}

C[indexC] = Ctemp;
```

32 global loads

(a) Initial Version

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

  // load input tile elements
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

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

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

2 global loads, 2 shared memory stores

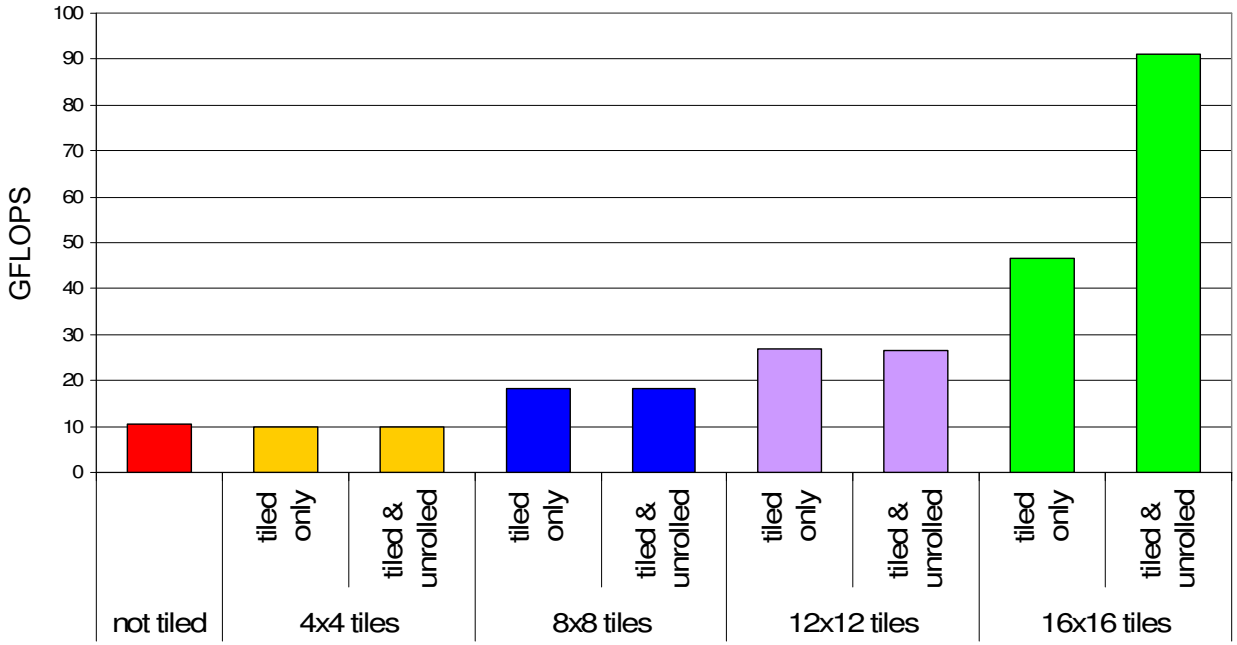
32 shared memory loads

(b) Tiled Version

Developers must correctly manage data locality.

Thread Grouping

- Each memory is specialized for certain access patterns
- Efficient global memory access is linked to tile size
- For good bandwidth utilization, accesses should be aligned and consist of 16 contiguous words



Further Improvement

- Reduce non-computation instructions
  - Loop unrolling
  - Change threading granularity to eliminate redundancy
- TLP vs. per-thread performance: prefetching, register tiling, traditional optimizations
- Difficult to estimate performance without performing the optimization and executing the kernel

Unrolling

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

  // load input tile elements
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

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

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

(b) Tiled Version

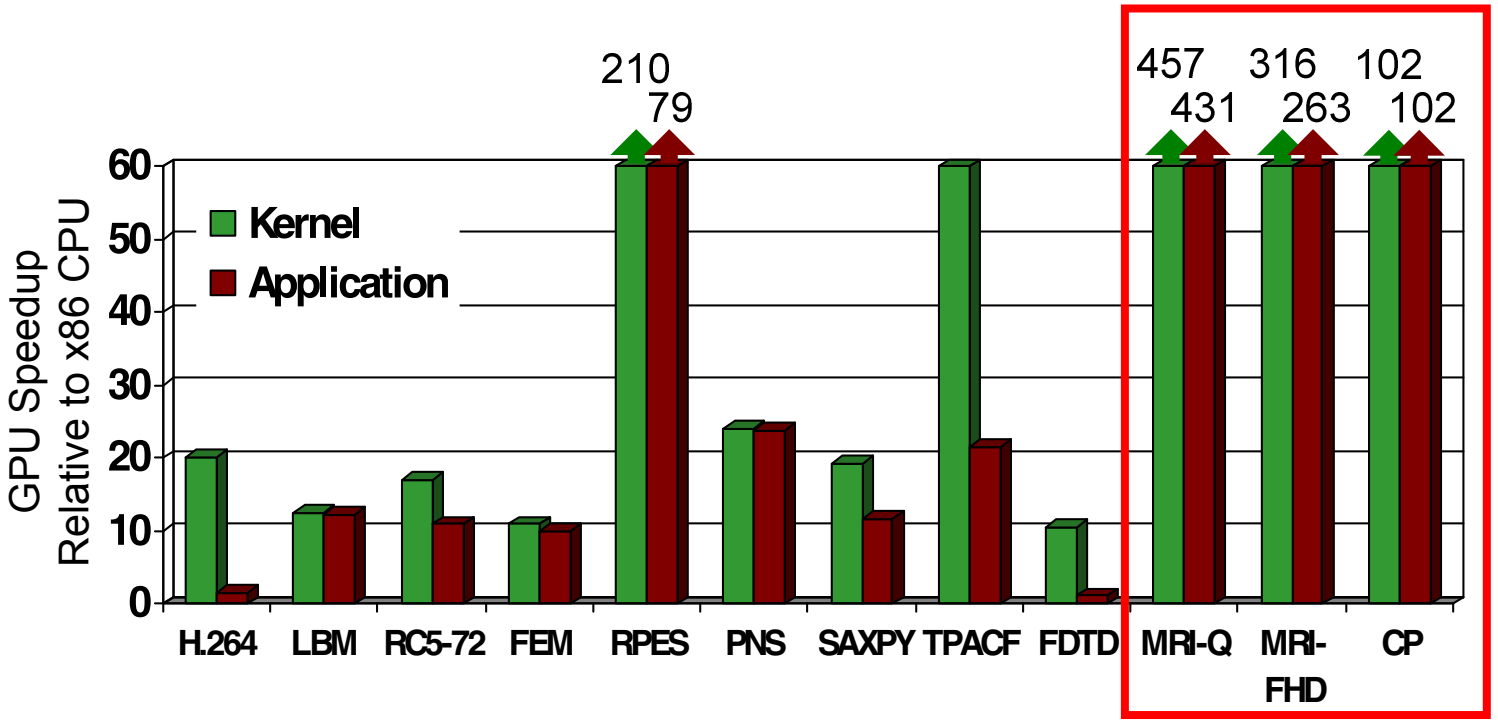
```
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  // load input tile elements
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  // compute results for tile
  Ctemp +=
     As[ty][0] * Bs[0][tx];
...
  Ctemp +=
     As[ty][15] * Bs[15][tx];

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

(c) Unrolled Version

Does this use more registers?

Removal of branch instructions and address calculations

Parboil: Speedup of GPU-Accelerated Functions

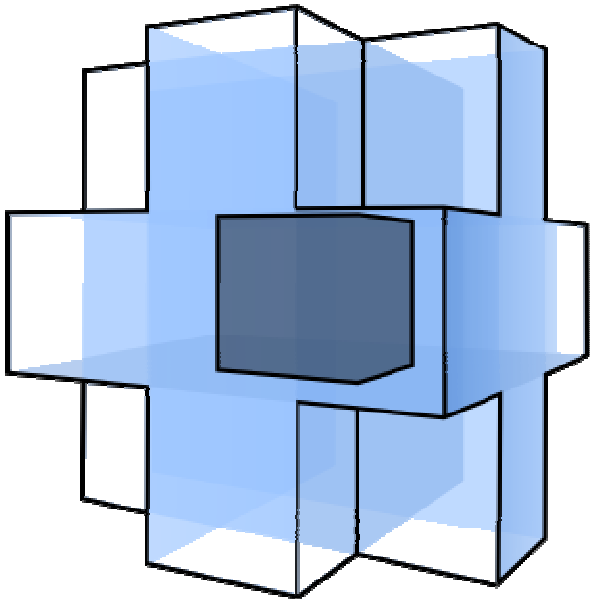


Most of these required 20+ different configurations

Speedup changes the usage model!

LBM Fluid Simulation (from SPEC CPU 2006)

- Simulation of fluid flow in a grid
- Synchronization required after each time step
- Can reduce bandwidth usage by caching in shared memory
- But can hold data for only 200 grid cells – global memory latency is exposed



Flow through a cell (dark blue) is updated based on its flow and the flow in 18 neighboring cells (light blue).

Conclusions

- Naïve mapping to GeForce 8800 does not always buy significant performance
- By following the presented principles, 10X or more performance advantage over a single-core CPU can be achieved
- Remaining speedup involves using specialized resources or trading off use of resources

Program Optimization Space Pruning for a Multithreaded GPU

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

Center for Reliable and High-Performance Computing
University of Illinois at Urbana-Champaign

Optimization

- First order principles [PPoPP '08]
  - Enough threads for TLP
  - Little or no inter-thread communication
  - Good use of on-chip memory to reduce bandwidth pressure
  - Thread grouping to optimize SIMD & memory banking
- Second order – balancing instruction stream efficiency and hardware utilization

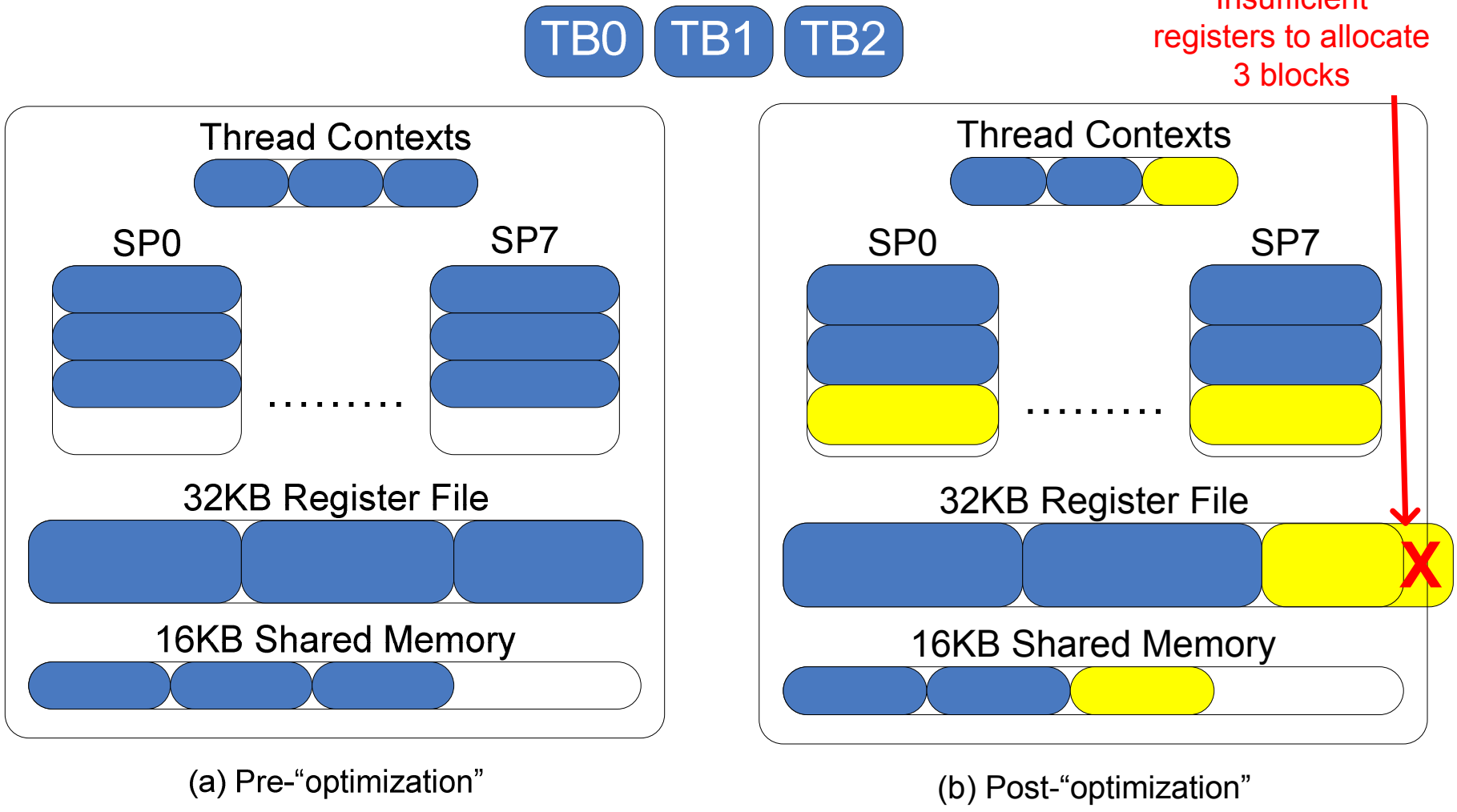
Examples of Optimizations

- Instruction stream efficiency
  - Loop unrolling
  - TLP granularity changes
  - Traditional optimization
- Hardware utilization
  - Prefetching
  - ILP-extraction transformations
  - Scheduling for MLP
  - Increasing TLP

Discontinuous Optimization Space

- Begin with a version with 3 thread blocks per SM
  - 256 threads per thread block, 10 registers per thread
  - 3 * 256 * 10 = 7680 total registers
- Perform an optimization that uses an additional register per thread
  - 3 * 256 * 11 = 8448 > 8k!
  - Performance per thread has increased
  - But this version has only 2 thread blocks per SM
- Is the optimized version better or worse?

Resource Allocation Example

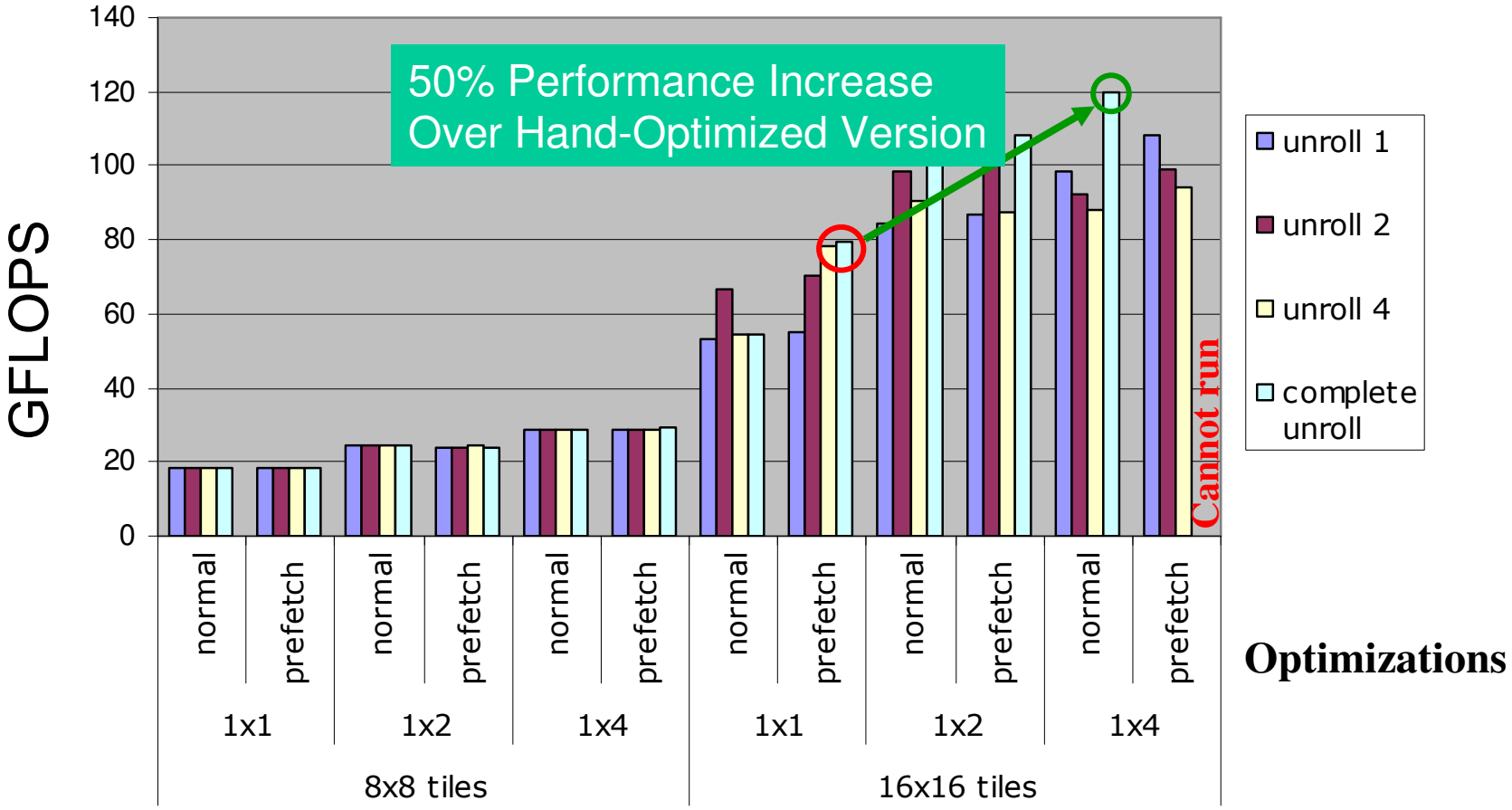


Increase in per-thread performance, but fewer threads:
Lower overall performance in this case

How Close Are We to Best Performance?

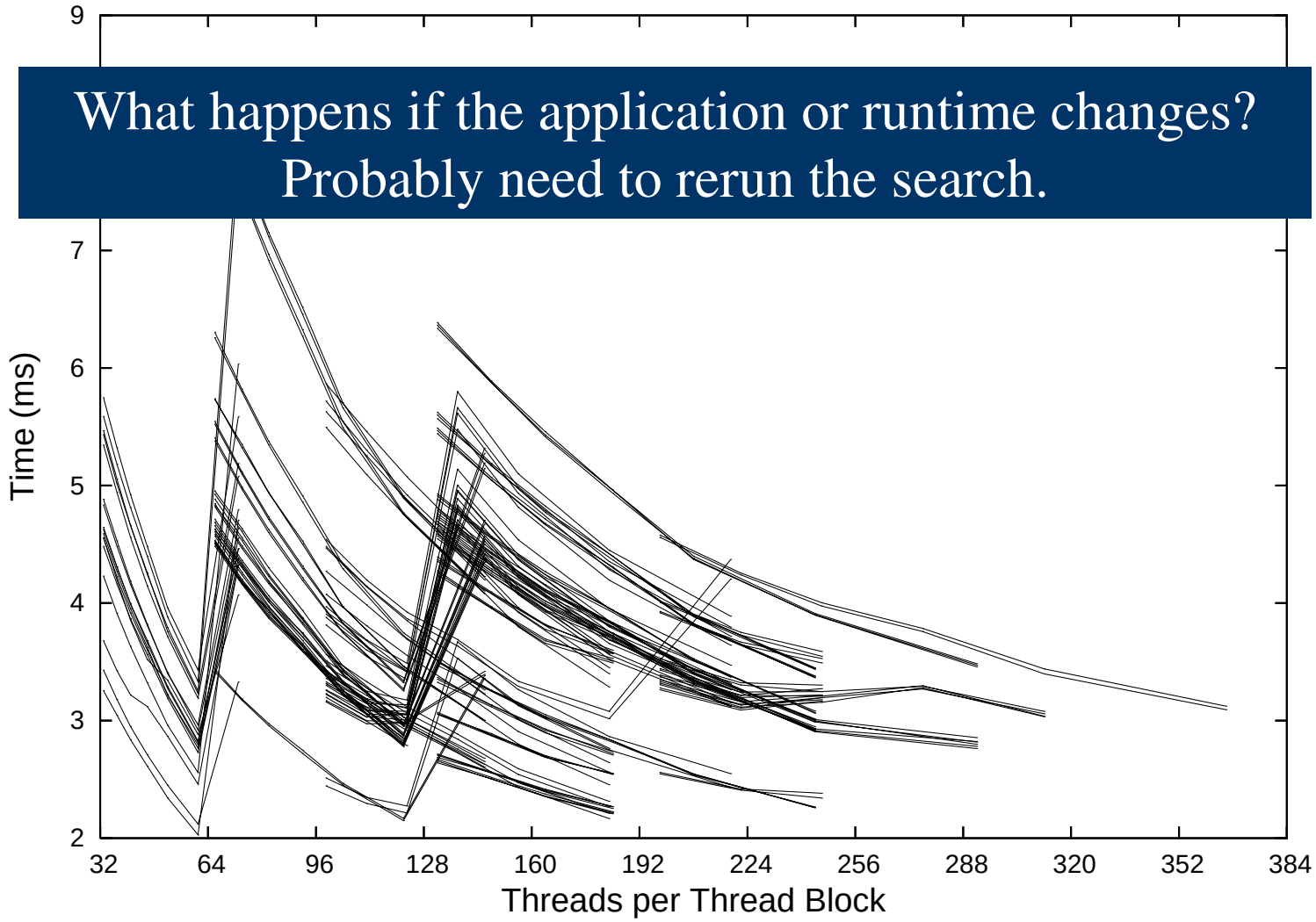
- Investigated applications with many optimizations
- Exhaustive optimization space search
  - Applied many different, controllable optimizations
  - Parameterized code by hand
- Hand-optimized code is deficient
  - Generally >15% from the best configuration
  - Trapped at local maxima
  - Often non-intuitive mix of optimizations

Matrix Multiplication Space



SAD Thread Block Size

What happens if the application or runtime changes?
Probably need to rerun the search.



Solution: Evaluate All Likely Configurations

- Begin with a complete, parameterized optimization space per kernel
  - Tile size and shape
  - Thread block size, work per thread block
  - Other optimizations, such as manual register spilling
- Compile all configurations to find resource usage
- Generate metric values for each configuration
- Prune space using a Pareto-optimal curve

Metrics for the GeForce 8800

How efficient is our instruction stream?

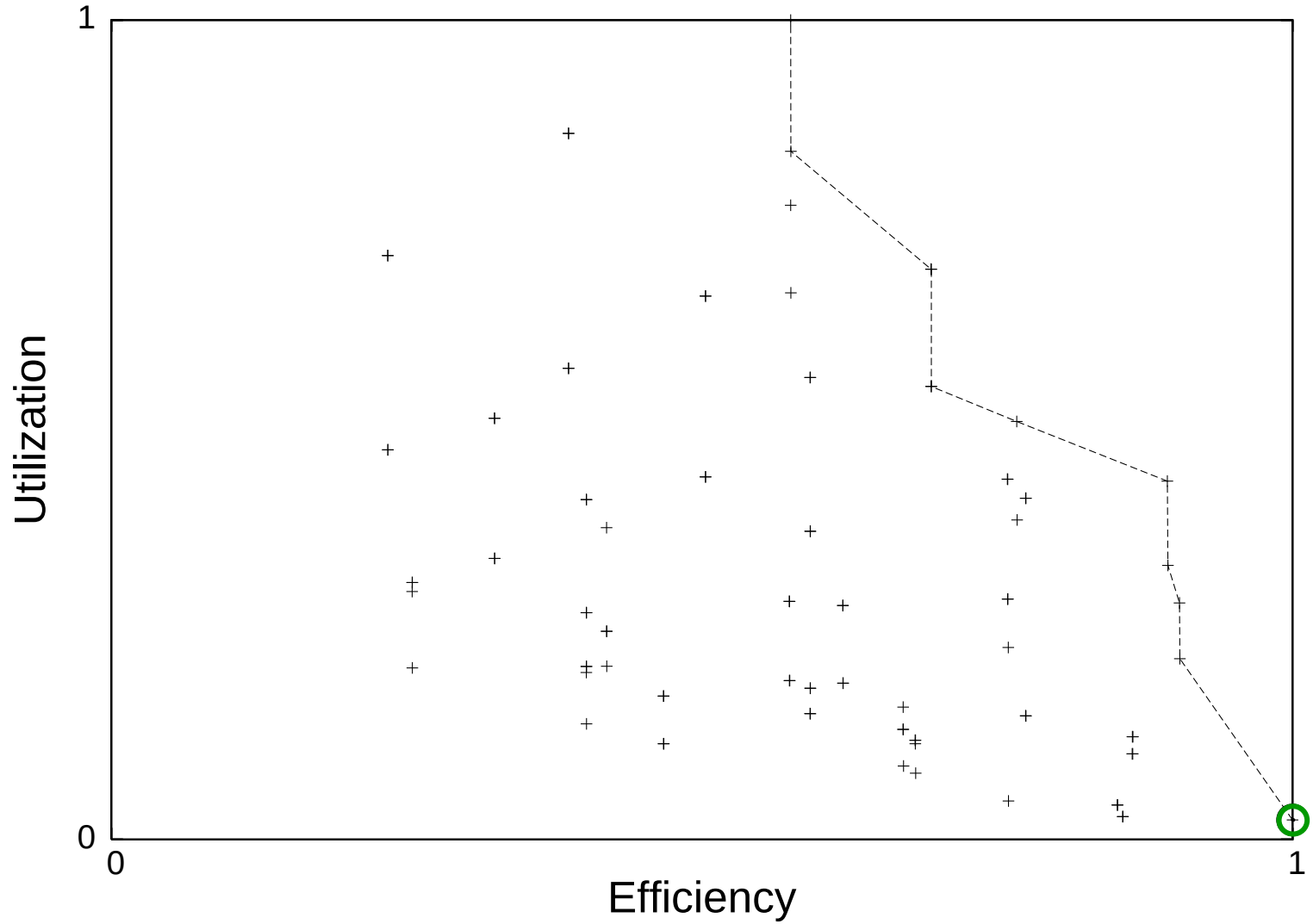
$$Efficiency = \frac{1}{Instr * Threads}$$

Can we keep the processors utilized?

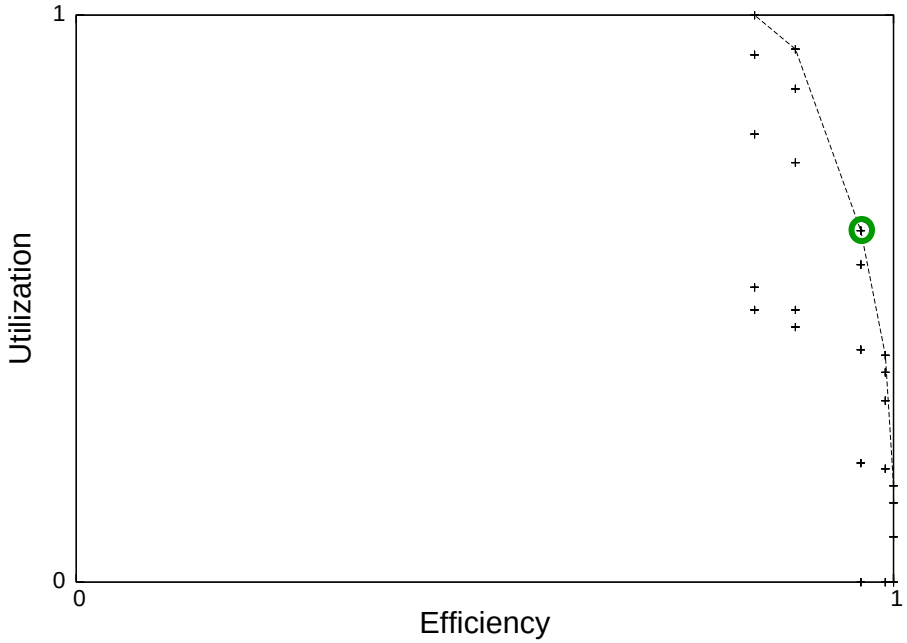
$$Utilization = \frac{Instr}{Regions}\left[\frac{W_{TB} - 1}{2} + (B_{SM} - 1)(W_{TB})\right]$$

These capture first-order effects of applications.

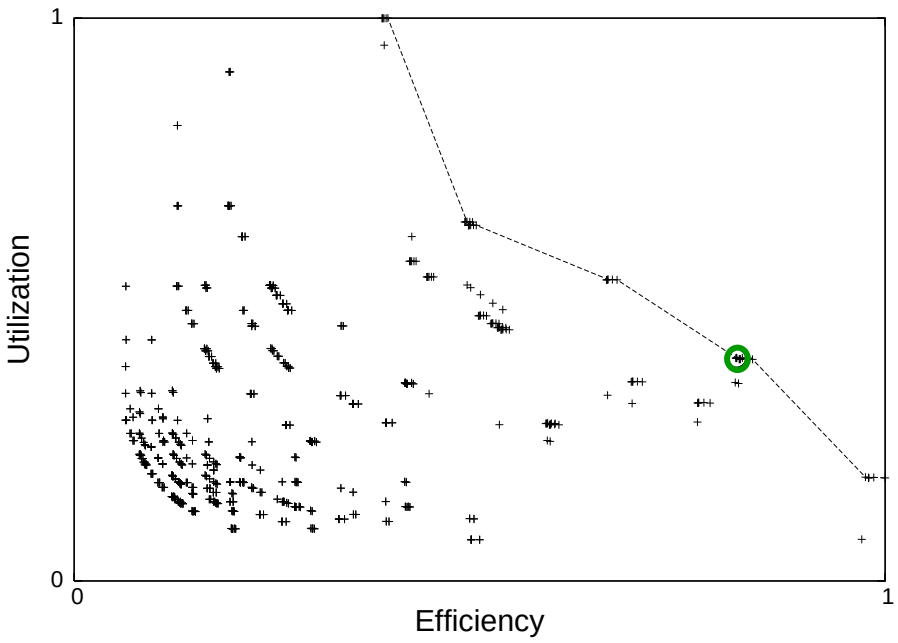
Matrix Multiplication Pareto Plot



MRI-FHD and SAD Plots



MRI-FHD



SAD

Space Reduction

Kernel	Total Configs	Total Eval	Selected Configs	Space Reduc.	Time Reduc.	Selected Perf.
MM	93	363.3 s	11	88%	87%	100%
CP	38	159.5 s	10	74%	73%	100%
SAD	908	7.677 s	16	98%	98%	100%
MRI-FHD	175	771.9 s	30	77%	73%	100%

Summary and Conclusion

- Performance tuning involves tradeoffs between instruction efficiency and HW utilization
- Manual, iterative optimization usually leaves performance on the table
- Reoptimization needed for new hardware, application, or runtime
- Pareto-based pruning removes up to 98% of the optimization space, still finds best configuration
- Ryoo Ph.D. dissertation contains further work: http://www.crhc.uiuc.edu/IMPACT.