Towards automatic parallelization and auto-tuning of affine kernels for GPUs

M. Baskaran U. Bondhugula S. Krishnamoorthy
J. Ramanujam A. Rountev P. Sadayappan

The Ohio State University
Louisiana State University

Introduction

- Emergence of many-core architectures
  - High computation power
  - e.g. GPUs
- Development of high-performance codes for such architectures – Non-trivial!
- CUDA parallel programming model for NVIDIA GPUs
  - Good abstraction of the underlying architecture
  - Not straight-forward to write a high-performance CUDA code

Introduction

- Optimizations needed to address architectural challenges
  - Memory access model
  - Granularity and levels of parallelism in architecture
- **Solution: Two-fold**
  - **Compiler infrastructure to automatically generate efficient parallel program structures;**
  - **Use model-driven empirical search for tuning**
- Framework:
  - PLuTo compiler framework [CC 08, PLDI 08] recently developed for gen-purpose multicore: Sequential C to OpenMP Parallel Tiled Code
  - Develop a framework to automatically generate parallel CUDA code

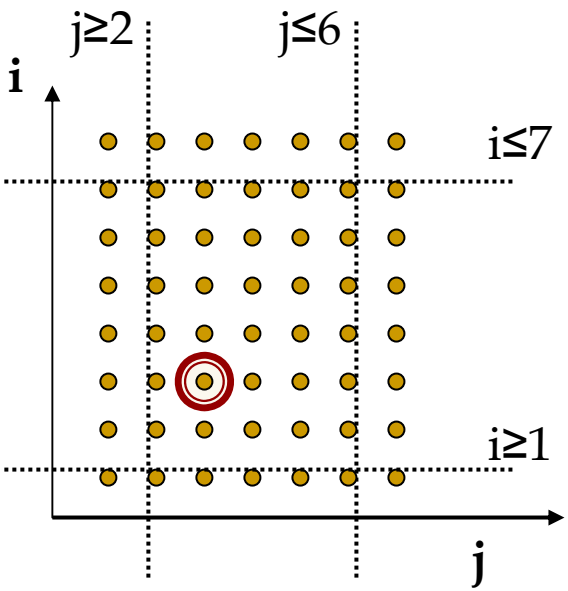
Polyhedral Model

- An algebraic framework for representing affine programs – statement domains, dependences, array access functions – and affine program transformations
- Regular affine programs
  - Dense arrays
  - Loop bounds – affine functions of outer loop variables, constants and program parameters
  - Array access functions - affine functions of surrounding loop variables, constants and program parameters

Polyhedral Model

```
for (i=1; i<=7; i++)
  for (j=2; j<=6; j++)
    S1: a[i][j] = a[j][i] + a[i][j-1];
```

$$\Phi_S(\vec{x_S}) = C_S \cdot \begin{pmatrix} \vec{x_S} \\ n \\ 1 \end{pmatrix}$$

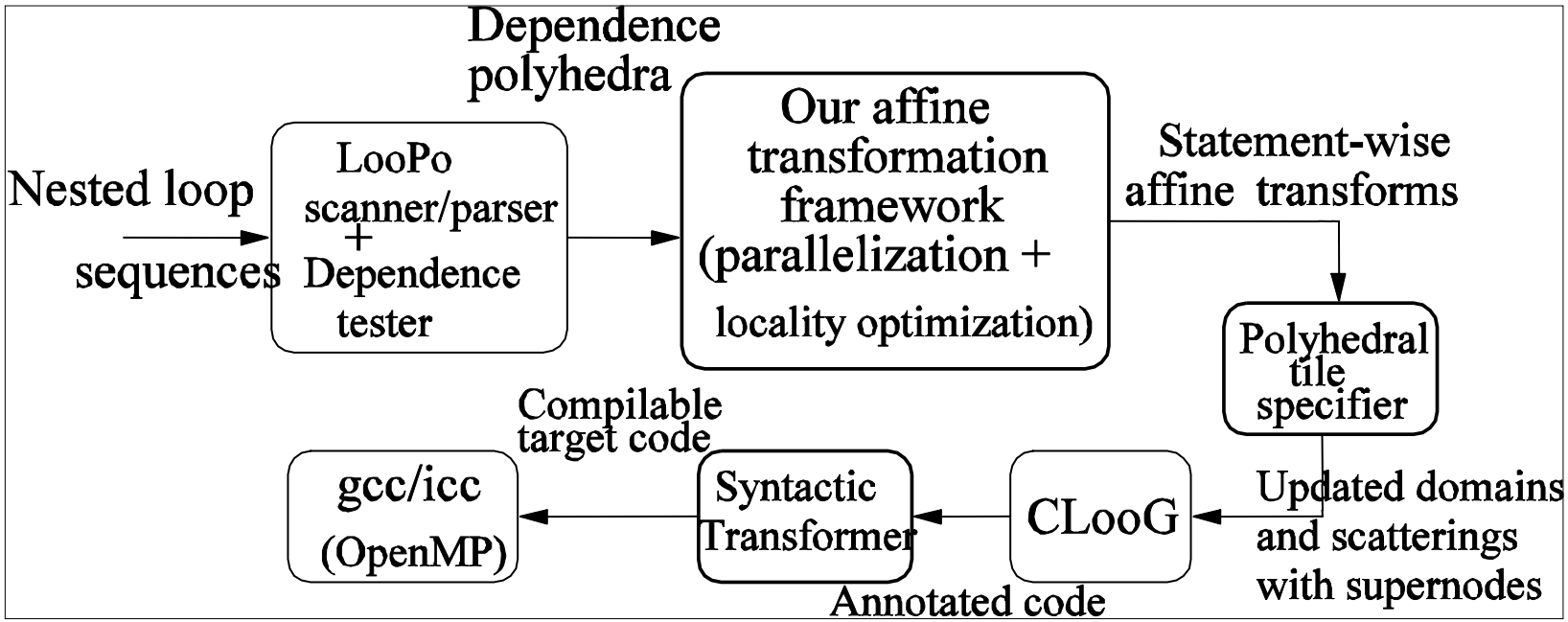


$$\vec{x_{S1}} = \begin{pmatrix} i \\ j \end{pmatrix}$$

$$F_{1a}(\vec{x_{S1}}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

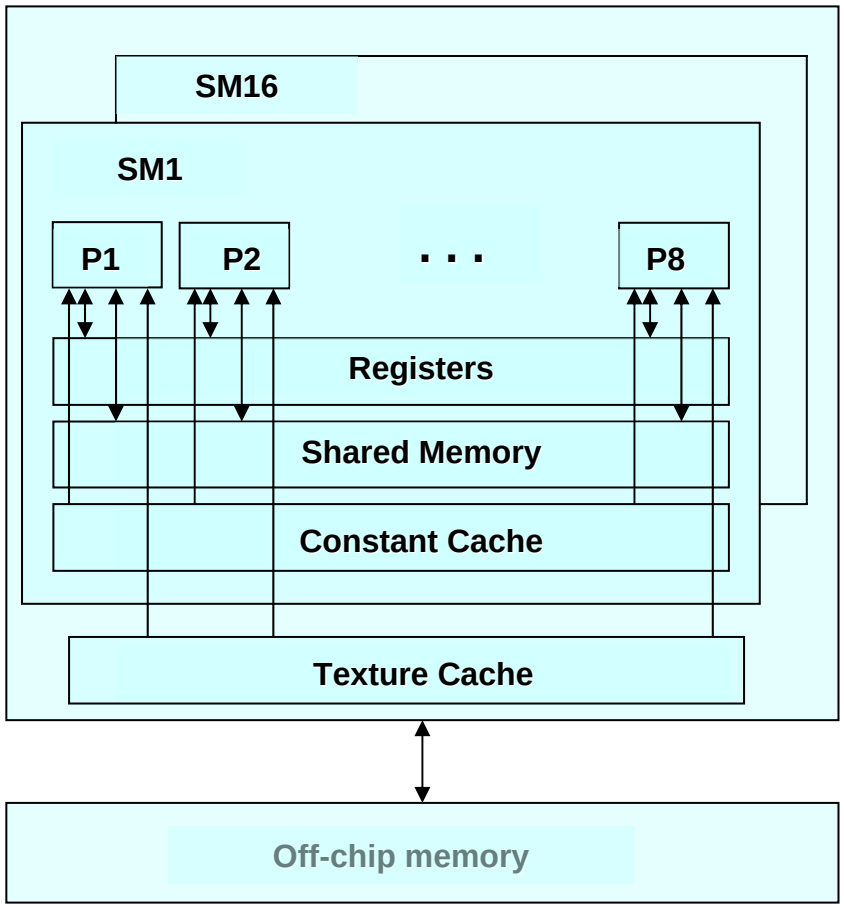
$$I_{S1} = \begin{pmatrix} 1 & 0 & -1 \\ -1 & 0 & 7 \\ 0 & 1 & -2 \\ 0 & -1 & 6 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \geq \vec{0}$$

PLuTo Framework



- Available at
  http://sourceforge.net/projects/pluto-compiler

NVIDIA GPU Architecture



- Two levels of parallelism
  - Threads (Processor cores)
    - Grouped into SIMD warps
  - Thread blocks (Multiprocessors)
- Various memory units
  - Different memory access model
  - Cache and local store hierarchy
- Partitioning (e.g. registers) and sharing of resources (e.g. shared memory)

Performance Characterization of NVIDIA GeForce 8800 GTX

- Get insights into optimizations to be addressed by a compiler framework
- Characterize key features of the machine architecture and their impact on different strategies:
  - Global memory access
  - Shared memory (scratchpad) access
  - Concurrency and register pressure

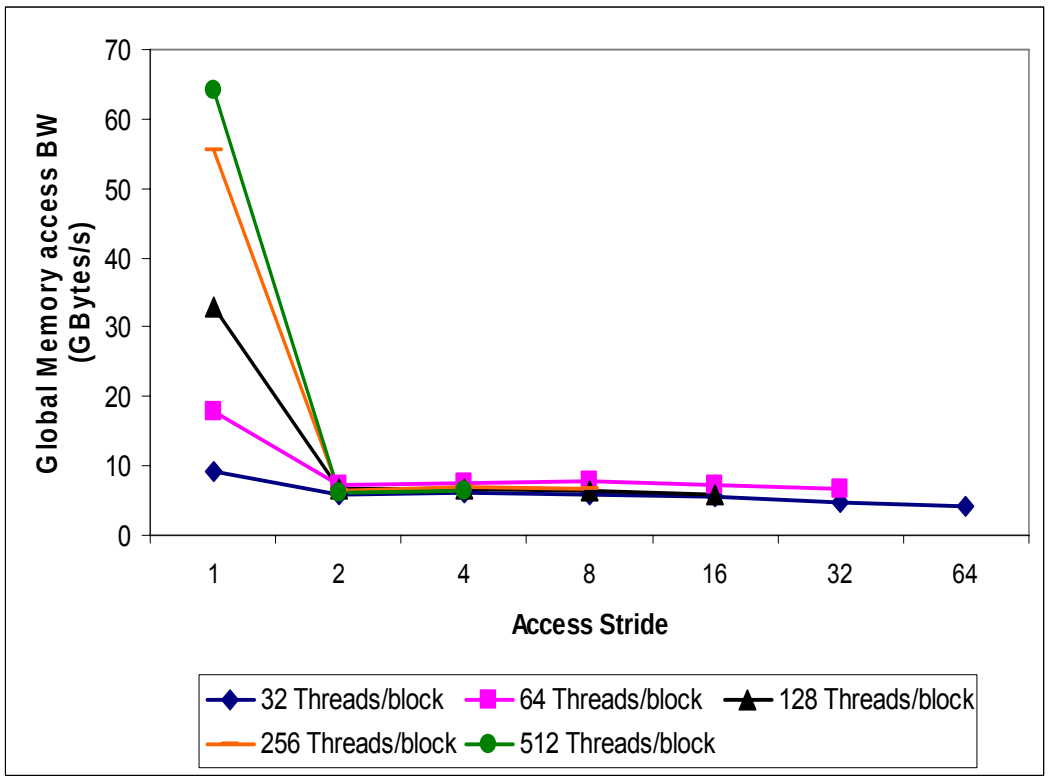
Global Memory Access-1

- Measured memory read bandwidth for
  - Different data sizes, N
  - Blocked and cyclic distribution of data access among the threads of a single thread block

N	Block (GB/sec)	Cyclic (GB/sec)
2048	4.11	22.91
4096	4.78	37.98
8192	5.11	48.20
16384	5.34	56.50
32768	6.43	68.51

Global Memory Access-2

- Measured memory read bandwidth for
  - Different strides (1 .. 64)
  - Different number of threads per thread block (32 ... 512)



Global Memory Access

- Cyclic access has much higher bandwidth
- Hardware optimization called **global memory coalescing**
  - Access from consecutive threads of a (half) warp to consecutive locations are coalesced
  - Base address of (half) warp aligned to 4, 8 or 16 bytes

Optimizing Global Memory Access

- Determine the extent of reuse of arrays
- For arrays with sufficient reuse
  - Copy from global memory to shared memory [PPoPP 08]
- For arrays with no reuse
  - Find affine transformations enabling global memory coalescing
  - If no suitable affine transformation enabling global memory coalescing
    - Copy to shared memory with possible global memory coalescing

Optimizing Global Memory Access

- To enable global memory coalescing for an array reference in a statement
  - Iterations accessing adjacent elements of an array (along the fastest varying dimension) executed at the same time point
- Enforced by the time schedule adjacency constraint

$$\forall x_s \in D_s, \forall y_s \in D_s \text{ s.t. } F_{szr}(\vec{x}_s) + \begin{pmatrix} 0 \\ \dots \\ 1 \end{pmatrix} = F_{szr'}(\vec{y}_s) \;\; \theta_s(\vec{x}_s) = \theta_s(\vec{y}_s)$$

Optimizing Global Memory Access

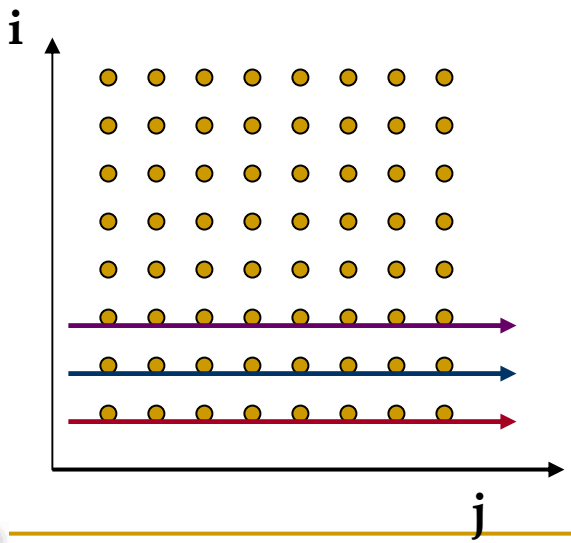
- To enable global memory coalescing for an array reference in a statement
  - Iterations accessing adjacent data elements of an array executed by adjacent threads
- Enforced by the space partition adjacency constraint

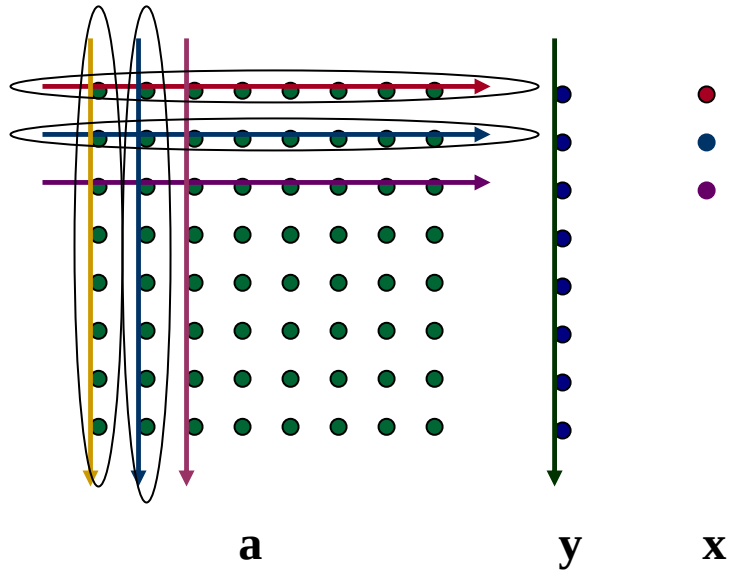
$$\forall x_s \in D_s, \forall y_s \in D_s \text{ s.t. } F_{szr}(\vec{x}_s) + \begin{pmatrix} 0 \\ \dots \\ 1 \end{pmatrix} = F_{szr'}(\vec{y}_s) \;\; \pi_s(\vec{x}_s) = \pi_s(\vec{y}_s) + 1$$

Optimizing Global Memory Access

tmvkernel:

```
for (i=0;i<n;i++) {
S: x[i]=0;
   for (j=0;j<n;j++)
Q:    x[i]+=a[i][j] * y[j];
}
```





$$\forall x_s \in D_s, \forall y_s \in D_s \text{ s.t. } F_{szr}(\vec{x}_s) + \begin{pmatrix} 0 \\ \dots \\ 1 \end{pmatrix} = F_{szr'}(\vec{y}_s) \;\; \theta_s(\vec{x}_s) = \theta_s(\vec{y}_s)$$

$$\forall x_s \in D_s, \forall y_s \in D_s \text{ s.t. } F_{szr}(\vec{x}_s) + \begin{pmatrix} 0 \\ \dots \\ 1 \end{pmatrix} = F_{szr'}(\vec{y}_s) \;\; \pi_s(\vec{x}_s) = \pi_s(\vec{y}_s) + 1$$

Experimental Evaluation

Performance comparison (in GFLOPS) of mv kernel

N	Direct Global	Copied to Shared
4K	0.43	5.61
5K	0.48	5.79
6K	0.35	6.04
7K	0.30	5.78
8K	0.24	5.52

Shared Memory Access

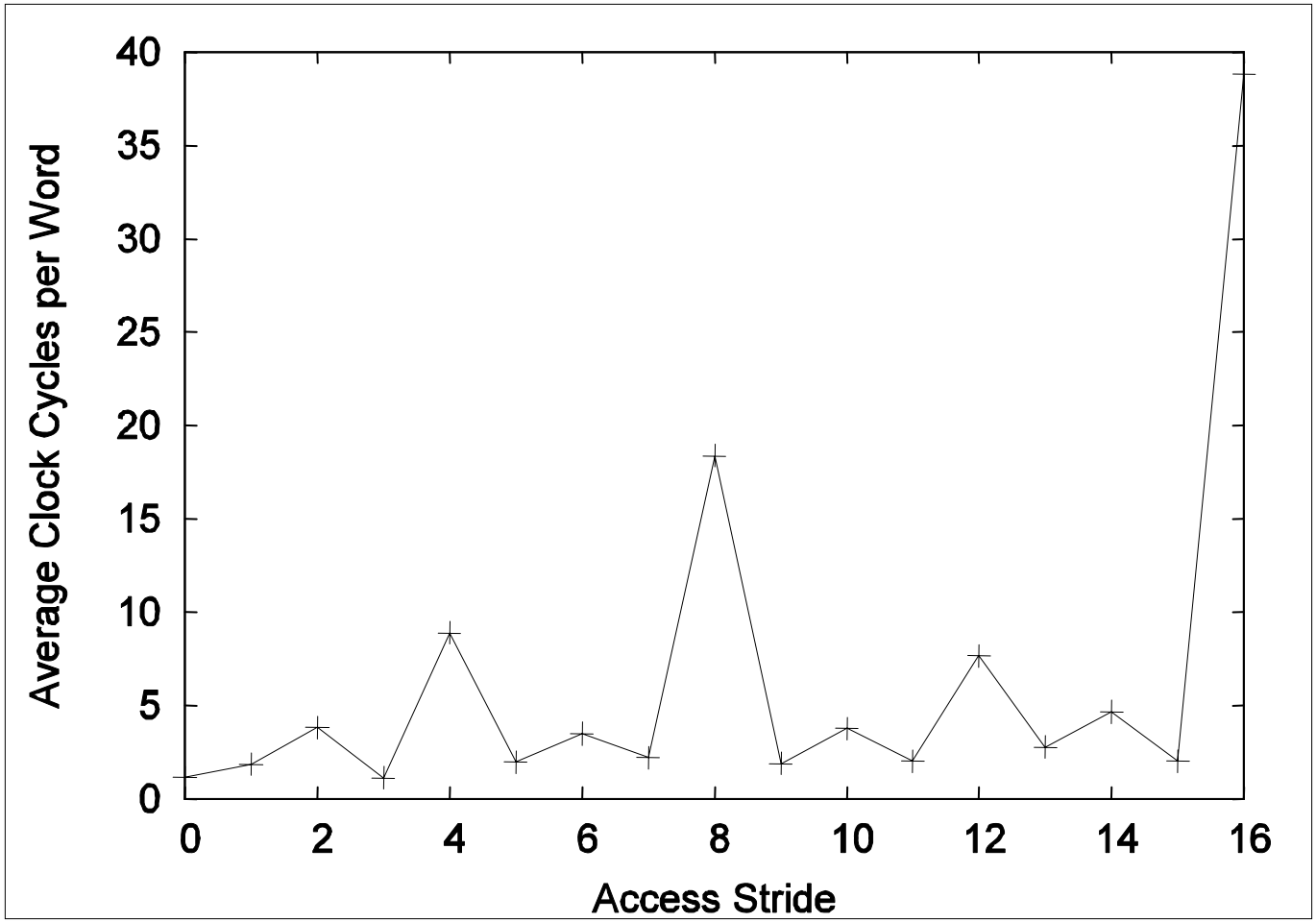
- Shared memory organized into banks
  - 16 banks in NVIDIA 8800 GTX
  - Successive 32-bit words in successive banks
- Bank conflicts in shared memory
  - n threads access different address in same bank – n sequential requests (n-way conflict)
  - Bandwidth of shared memory access *inversely proportional* to the degree of bank conflicts
- Goal: To minimize shared memory bank conflicts

Optimizing Shared Memory Access

- Strategy to minimize bank conflicts during shared memory access
  - Pad the arrays copied into shared memory
- Degree of bank conflicts =

  gcd (stride of array access across threads of a half warp, number of bank modules)
- Cost of accessing a word in shared memory
  - Linear function of degree of bank conflicts
- Find padding factor that minimizes the cost considering all references to the array

Shared Memory Access



Scratchpad: Challenges

- Effective management of on-chip scratchpads in multi-core architectures
  - Utilize limited capacity of scratchpad
  - Optimize data movement
- Effective computation mapping in many-core architectures with multiple levels of parallelism
  - Exploit available parallelism
  - Account for scratchpad capacity constraints

Data Management Issues

- Orchestration of data movement between off-chip global and on-chip scratchpad memory

- Need to make decisions on
  - What data elements to move in and out of scratchpad
  - When to move data
  - How to move data
  - How to access the data elements copied to scratchpad

Data Management Approach

1. Allocation of storage space (as arrays) in the scratchpad memory for local copies
2. Determination of access functions of arrays in scratchpad memories
3. Generation of code for moving data between scratchpad (local) and off-chip (global) memories

Overview: Data Management Approach

- Targeted at affine programs
  - Dense arrays
  - Loop bounds – affine functions of outer loop variables, constants and program parameters
  - Array access functions - affine functions of surrounding loop variables, constants and program parameters
- Developed using *polyhedral model*

Automatic Data Allocation

- Given a program block, identify the storage space needed for each non-overlapping accessed region of all arrays
  - Access functions of array references may be non-uniformly generated
- For some architectures (such as the NVIDIA GeForce GPU) supporting direct data access from off-chip memory
  - Estimate extent of reuse of data to determine whether or not to copy to scratchpad

Data Movement Code Generation

- Generation of loop structure
  - Scanning of polytopes (using CLooG - a tool for code generation) corresponding to data spaces of
    - read references: for moving data into scratchpad
    - write references: for moving data out of scratchpad
- Generation of loop body (data movement statement)
  - Copy from a location in scratchpad buffer to off-chip memory location or vice-versa

Experimental Evaluation

Performance comparison (in GFLOPS) of mv kernel

N	Non optimized Shared	Optimized Shared
4K	5.61	13.18
5K	5.79	13.87
6K	6.04	14.37
7K	5.78	13.86
8K	5.52	13.63

Parallelism vs Register Pressure

- Performance enhancing approaches
  - Reduction of number of loads/stores
  - Increase in ILP
  - Reduce dynamic instructions
    - Loop overhead reduction
- Well-known optimization: Loop unrolling
- Issues
  - Increased register pressure
  - Might reduce number of concurrent threads
    - Registers are partitioned among thread blocks

Parallelism vs. Register Pressure

- Higher thread-level parallelism needed to mask global memory access latency
  - Threads scheduled in an interleaved manner to mask global memory access latency
- Trade-off between
  - number of active concurrent threads
  - number of registers available for a thread in a thread block
- Problem: Register allocation cannot be managed by any external compilation framework
- Solution: Empirical evaluation

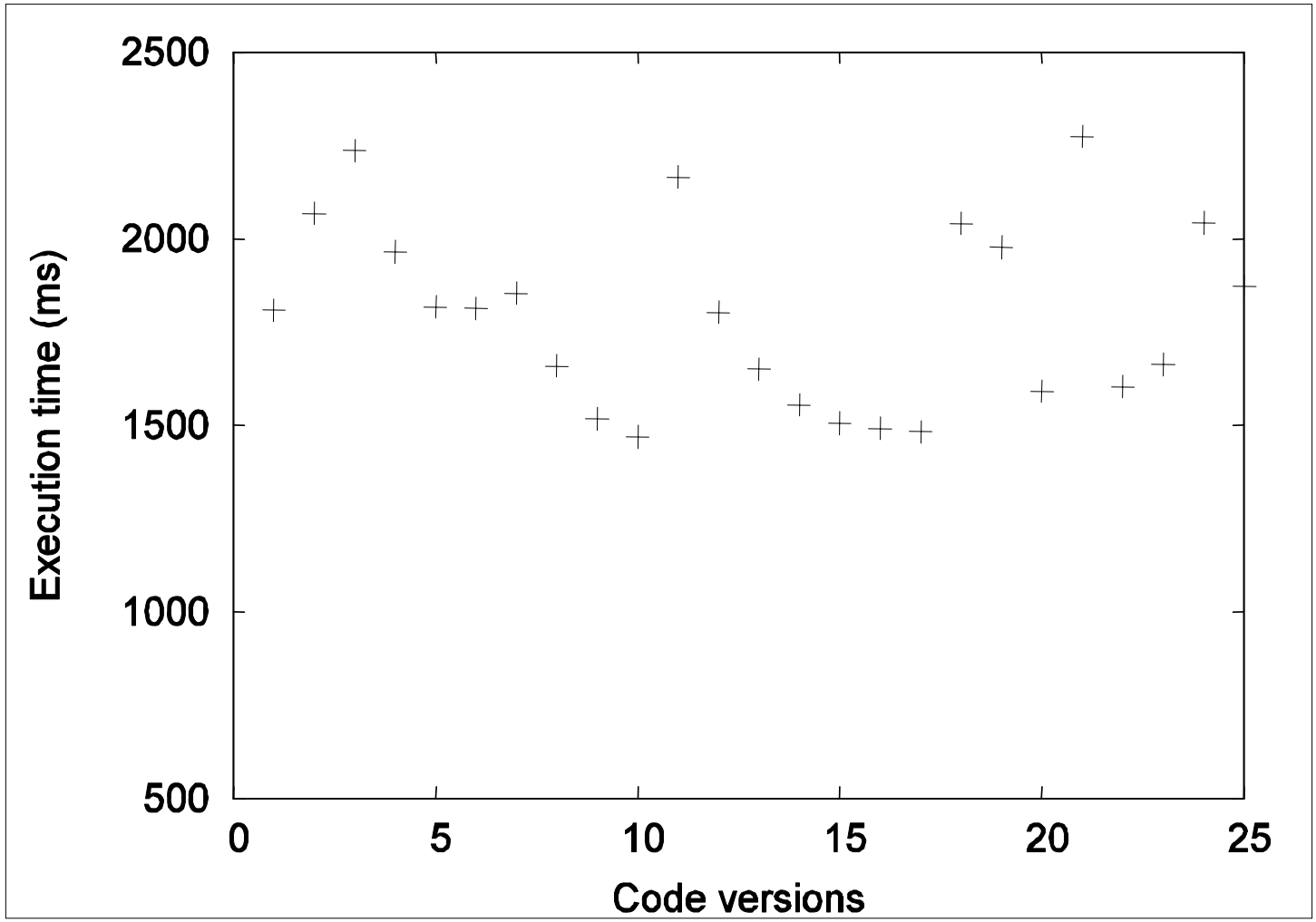
Model-driven Empirical Search

- Need for empirical search
  - Tight coupling
    - Program parameters – tile sizes, unroll factors
    - System parameters – threads, thread blocks
    - Resources - Number of registers, shared memory
  - Lack of control on registers (usage and allocation)
- Model to estimate number of loads/stores
  - Analytically in polyhedral model
  - Empirically using ptx code
- Register usage instrumentation
  - Empirically using cubin object code

Empirical search: MM kernel

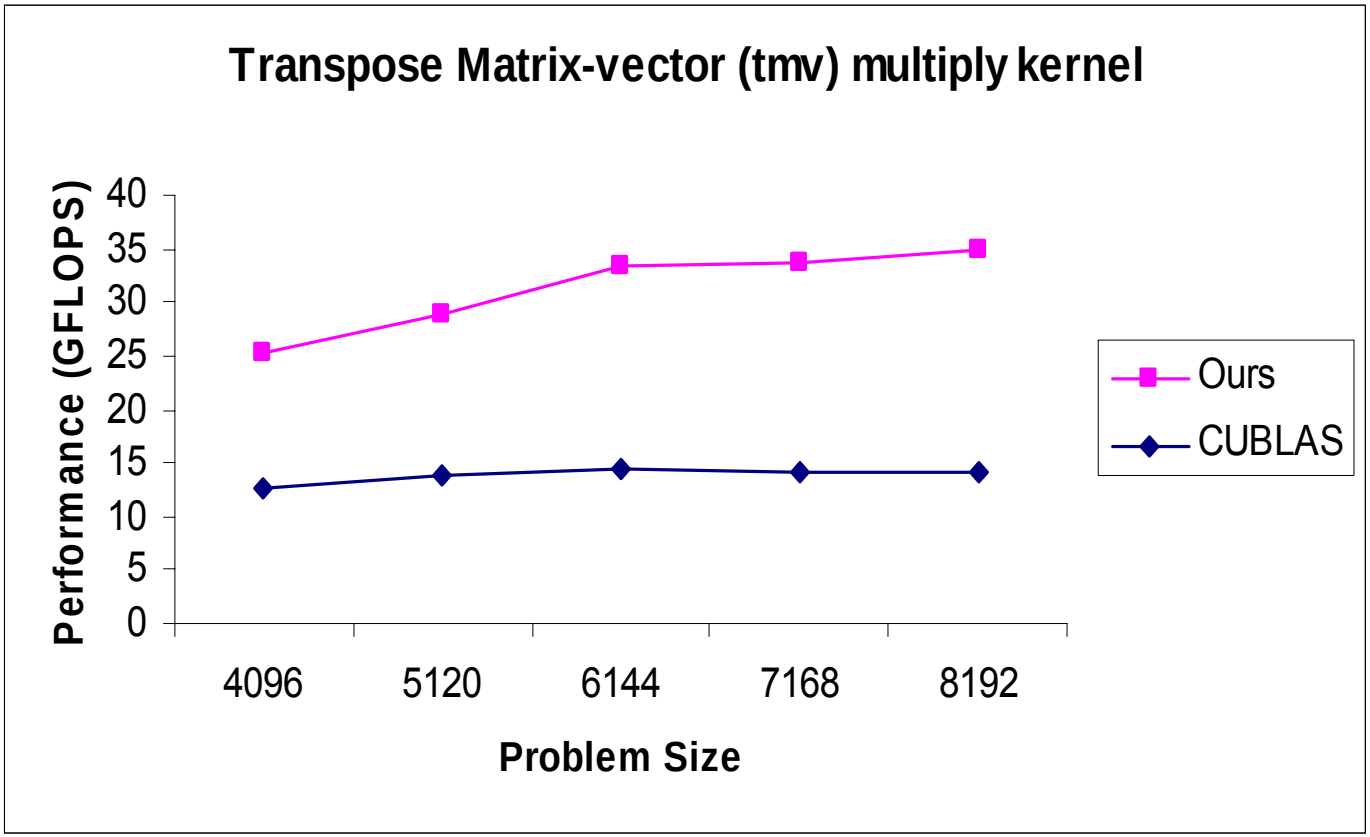
- Problem size: 4K x 4K (just fits in GPU DRAM)
- Number of thread blocks: 16, 32, 64
- Threads per thread block: 128, 256, 512
- For different load balanced versions that do not exceed shared memory limit, the global memory loads vary from $(4K)^3/2^7$ to $(4K)^3/2^4$.
- We considered versions with global memory loads from $(4K)^3/2^7$ to $(4K)^3/2^6$ and used different combinations loop unroll factors and register tiling

Empirical search: MM kernel



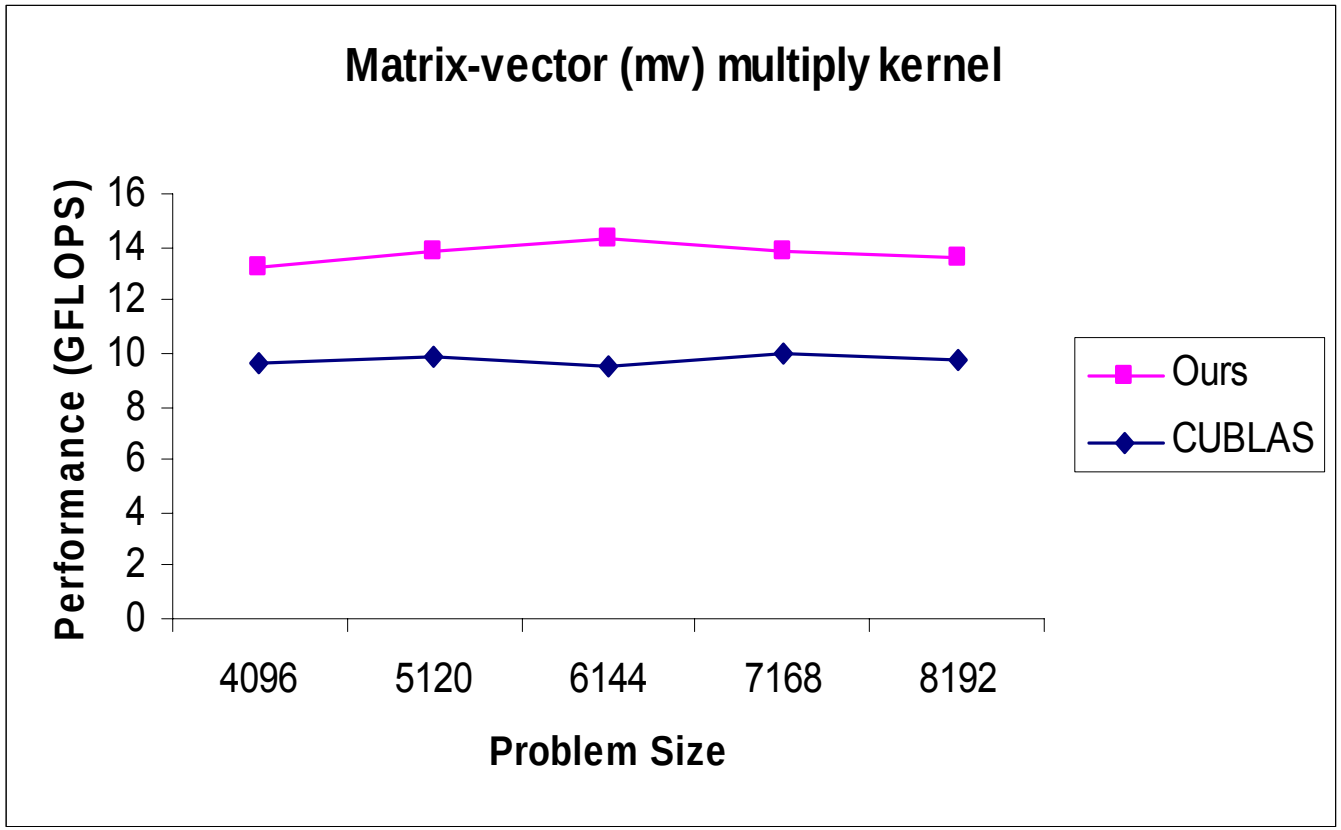
Experimental Evaluation

Performance of Matrix Kernels



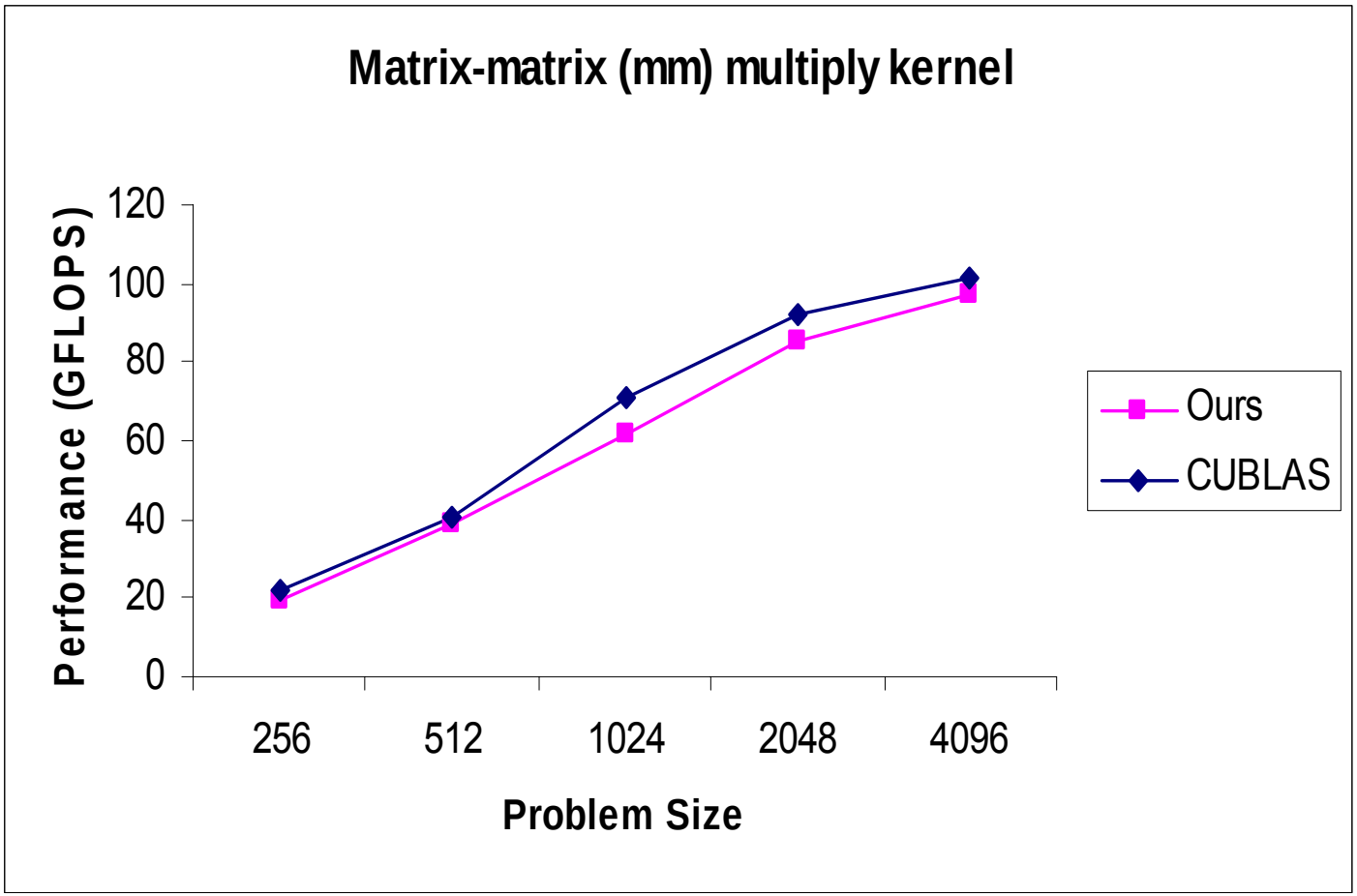
Experimental Evaluation

Performance of Matrix Kernels



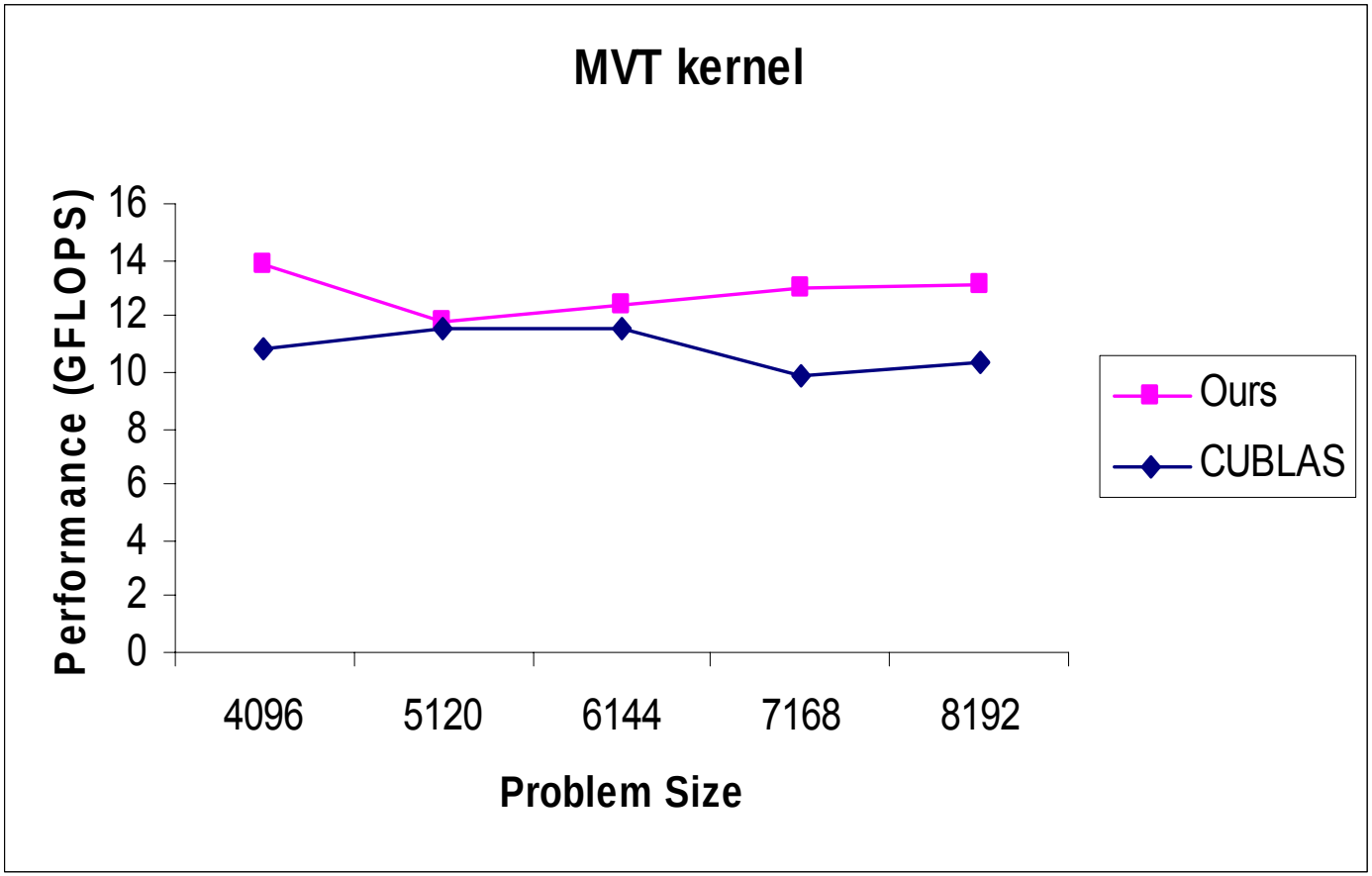
Experimental Evaluation

Performance of Matrix Kernels



Experimental Evaluation

Performance of Matrix Kernels



Related Work

- Earlier GPU works
  - Automatic generation of pixel shader operations from a high-level data-parallel language – Tarditi et al. [ASPLOS'06]
  - Stream processing – Brook, RapidMind, PeakStream
- Considerable work on developing specific optimized algorithms and libraries for GPUs
  - e.g. CUDPP – CUDA Data Parallel Primitives
- Very little work on general compiler optimization strategies for GPUs
  - Performance metrics to prune the optimization search space on a pareto-optimality basis - by Ryoo et al. [CGO'08]
  - Optimize data communication between CPU and co-processor – by Gelado et al. [ICS'08]

So far …

- Developed compiler optimizations to address key performance-influencing factors on NVIDIA GPUs
  - Enable global memory coalescing in the polyhedral model for regular programs
  - Reduce shared memory bank conflicts
  - Determine optimized program parameters (unroll factors, tile sizes) through a model-driven empirical search

Ongoing Work

- Automatic *thread-centric* CUDA Code Generation in Polyhedral Model
- Data layout reordering to enhance memory accesses at various levels

Thank You!