Locality-Aware Mapping of Nested Parallel Patterns on GPUs

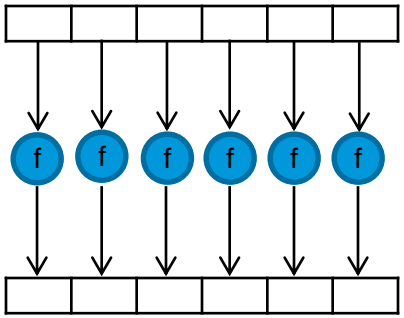
HyoukJoong Lee[*], Kevin Brown[*], Arvind Sujeeth[*], Tiark Rompf[†‡], Kunle Olukotun[*]

[*]Pervasive Parallelism Laboratory, Stanford University

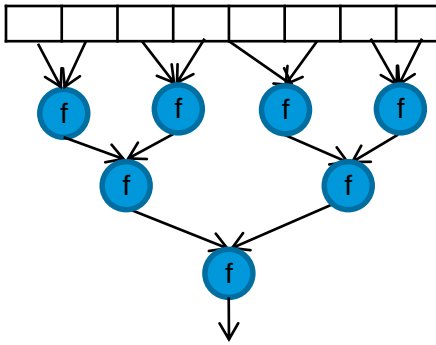
[†]Purdue University, [‡]Oracle Labs

High-level Languages for GPUs

- Provide higher productivity and portable performance
- Parallel patterns are becoming a popular abstraction for computations
  - map, reduce, filter, groupby, …
  - Supported by Copperhead, Lime, Accelerate, Thrust, ..
  - Provide high-level information on parallelism and internal communication
- Compilers often support a fixed mapping strategy for each pattern



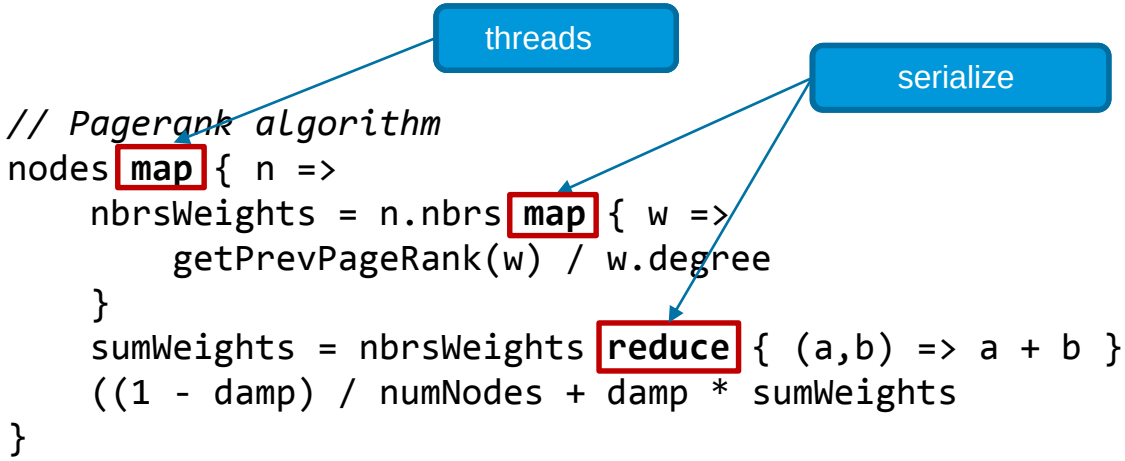
out = in.map(f)



out = in.reduce(f)

Challenges

- Parallel patterns are often nested in applications
  - > 70% apps in Rodinia benchmark contain kernels with nested parallelism
- Efficiently mapping parallel patterns on GPUs becomes significantly difficult when patterns are nested
  - Many factors to consider together (e.g., coalescing, divergence, dynamic allocations)
  - Large space of possible mappings



Existing Mapping Strategies

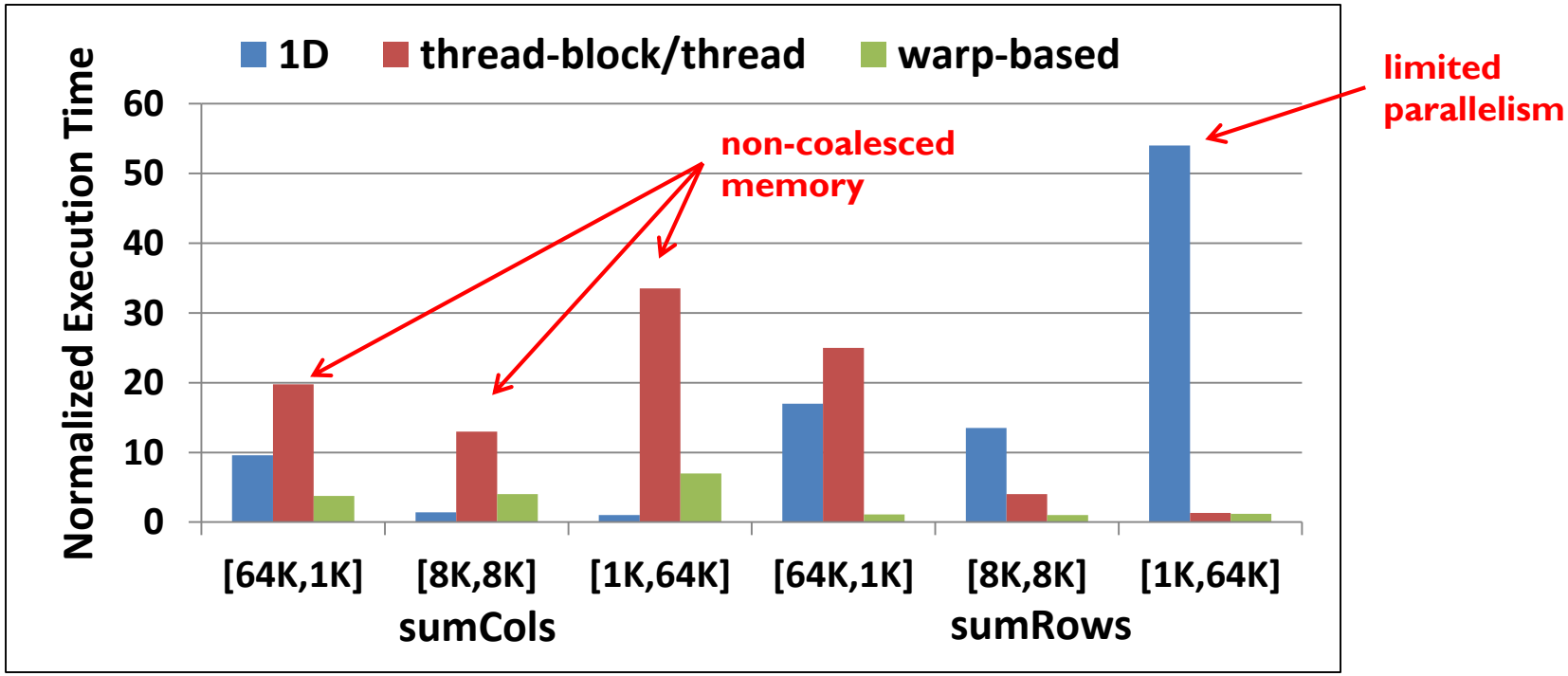
- 1D mapping
  - Only parallelize one of the loops (often either inner-most or outer-most)
  - Sequentially execute other loops
  - Default mapping strategies for many compilers
- Thread-block / thread mapping
  - Assign each outer loop iteration to a thread-block
  - Inner loop is parallelized by threads within a thread-block
  - *Bryan Catanzaro, et al. "Copperhead: Compiling an Embedded Data Parallel Language", PPoPP 2011*
- Warp-based mapping
  - Assign a warp (32 SIMD execution unit) to one or more outer loop iterations
  - Inner loop is parallelized by threads in a warp
  - *Sungpack Hong, et al. "Accelerating CUDA Graph Algorithms at Maximum Warp", PPoPP 2011*

Issues with Existing Mappings

```
m = Matrix.rand(nR,nC)
v = m.sumCols
```

```
map (i)
  reduce(j)
```

```
m = Matrix.rand(nR,nC)
v = m.sumRows
```

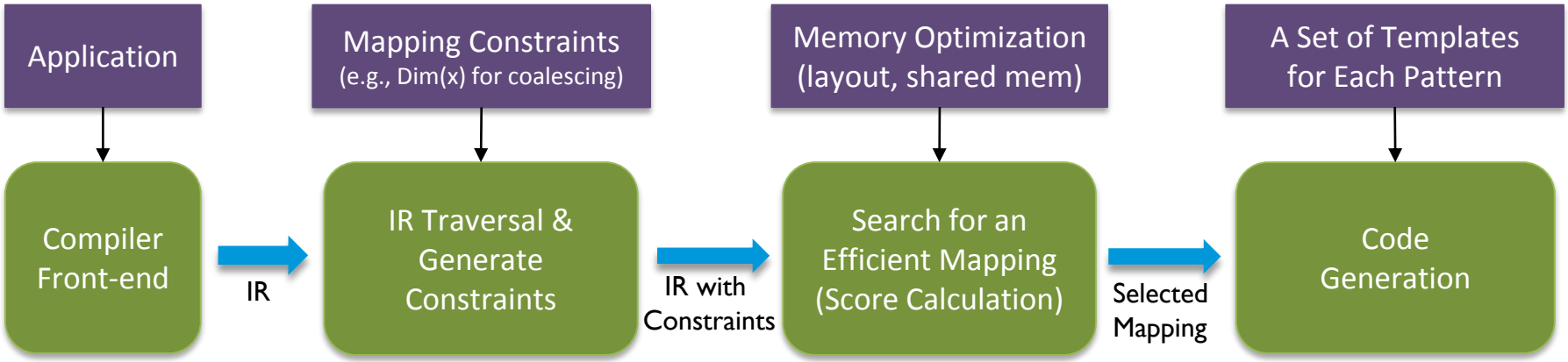


Compiler Framework for Multi-Dimensional Mapping

- Define Mapping Parameters
  - Flexible enough to cover existing mapping strategies

Logical Dimension: x, y, z, ..
Block Size: N
Degree of Parallelism (DOP): Span(n), Span(all), Split(k)

- Compiler Overview



Outline

- Introduction
- Input and Output of Mapping Analysis
  - IR and Mapping Parameters
- Search for an Efficient Mapping
  - Mapping Constraints and Scores
  - Dynamic Memory Optimization
- Evaluation
- Conclusion

Intermediate Representation (IR)

- Input to our compiler analysis
- Based on existing parallel pattern languages / data parallel languages
- Structured computations and data structures
  - Computations

Pattern	Example
map	`in map { e => e + 1 }`
zipwith	`inA zipWith(inB) { (eA,eB) => eA + eB }`

```
// Pagerank algorithm
nodes map { n =>
    nbrsWeights = n.nbrs map { w =>
        getPrevPageRank(w) / w.degree
    }
    sumWeights = nbrsWeights reduce { (a,b) => a + b }
    ((1 - damp) / numNodes + damp * sumWeights
}
```

- We implemented a data-parallel language around the IR

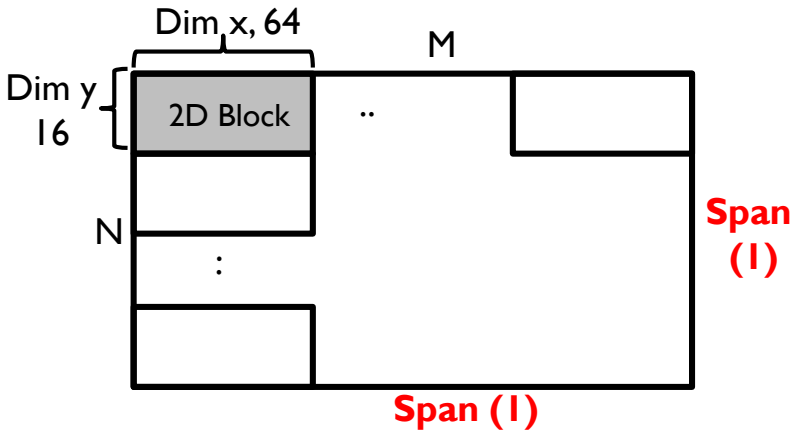
Mapping Parameters

- Result of our compiler analysis
- For each nest level, (Dimension, Block Size, Degree of Parallelism)

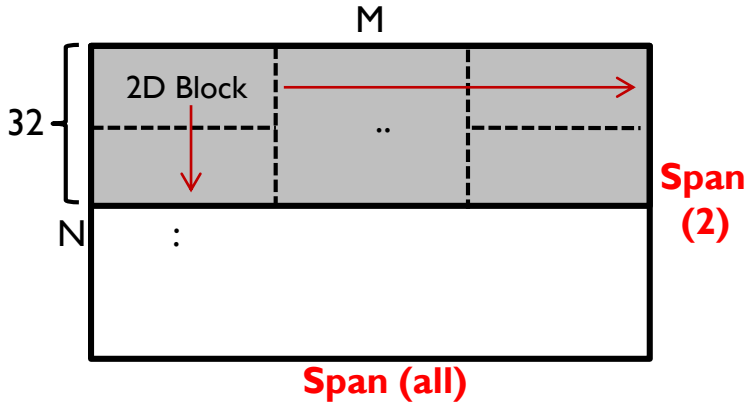
```
Pattern (I)        // Dim(Y), 16, Span(1)
    Pattern (J)    // Dim(X), 32, Span(all)
```

- Dimension
  - A logical dimension assigned to the index domain of a nest level
  - Compiler controls how indices in each dimension are mapped to HW threads
- Block size
  - Number of threads assigned for a given dimension
- Degree of Parallelism (DOP)
  - The amount of parallel computations enabled by a mapping
  - Controls how computations are assigned to threads
  - Span(n) and Split(k) decreases / increases DOP respectively

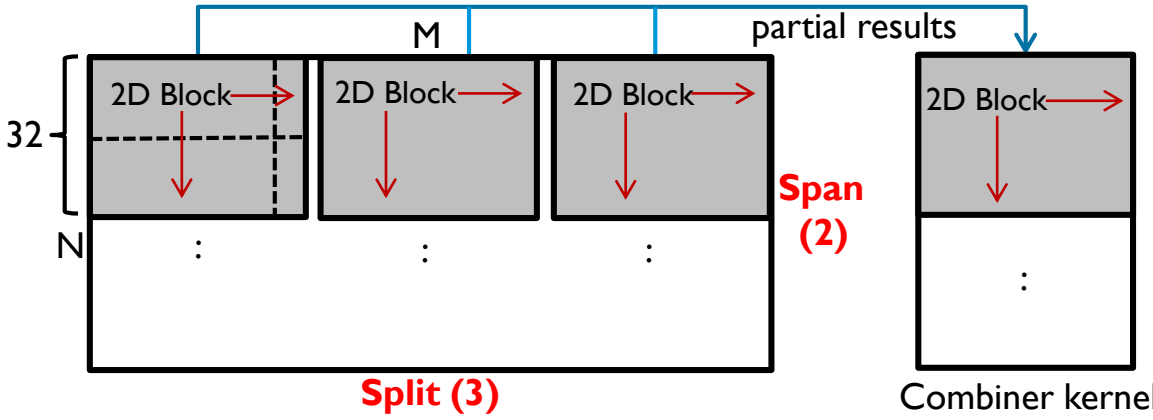
Degree of Parallelism (DOP)



(a) Span(1) on both dimensions



(b) Span(all) on Dim x and Span(2) on Dim y



(c) Split(3) on Dim x and Span(2) on Dim y, launch an additional combiner kernel

Comparison to Existing Mapping Strategies

- Thread-block / thread mapping (DOP: I * min(J, *MAX_BLOCK_SIZE*))

```
Pattern (I)      // assign a thread-block
  Pattern (J)    // threads (1024) in a block
```

```
Pattern (I)      // DimY, 1, Span(1)
  Pattern (J)    // DimX, 1024, Span(all)
```

- Warp-based mapping (DOP: I * min(J, *WARP_SIZE*))

```
Pattern (I)      // assign a warp
  Pattern (J)    // threads (32) in a warp
```

```
Pattern (I)      // DimY, 16, Span(1)
  Pattern (J)    // DimX, 32, Span(all)
```

- Flexible enough to cover existing mapping strategies
- More flexible than existing fixed strategies
- Provides a better view of similarities and differences between different mapping strategies

Outline

- Introduction
- Input and Output of Mapping Analysis
  - IR and Mapping Parameters
- Search for an Efficient Mapping
  - Mapping Constraints and Scores
  - Dynamic Memory Optimization
- Evaluation
- Conclusion

Mapping Constraints

- Prunes the mapping space
  - Dynamically generated while traversing the IR
- Constraints from common GPU optimizations (soft)
  - Maximize memory coalescing
  - Provide enough parallelism
  - Avoid thread divergence
- Constraints from GPU HW / programming model (hard)
  - Max number of threads per block
  - Synchronizations across thread-blocks is not available
- Characteristics of parallel patterns (local / global)
  - Pick the most conservative span type within the same nest level

Soft Constraints

- Each soft constraint has an intrinsic weight
  - Based on empirical study of their relative impact on performance
  - Multiplied by the number of times the code will be executed
    - Multiply by the pattern size, discount by the branching factor

```
Pattern1 with i in Domain(0,I) {
      array1D(i)          # weight: α*I
      Pattern2 with j in Domain(0,J) {
          array2D(i,j)    # weight: α*I*J
} }
```

  - Exact values less important than the relative orderings
    - Effectively prioritize constraints applied in the inner-most nest level
    - Prioritizes more important soft constraint within the level

- Soft constraints may conflict with each other

Search for an Efficient Mapping

PERVASIVE PARALLELISM LABORATORY PPL

Entire mapping space:
exponential to the loop nests (base |DimSet| ∗ |SizeSet| ∗ |SpanSet|)

- Score calculation based on soft constraints
  - Adds all the scores from satisfied soft constraints
  - For unknown information at compile time, assume default values
- Adjust DOP
  - Span(all) -> Split(k)
  - Span(1) -> Span(n)
- Detailed decisions can also be adjusted at runtime
  - Changes that can be made without changing the mapping structure (e.g., thread-block size)

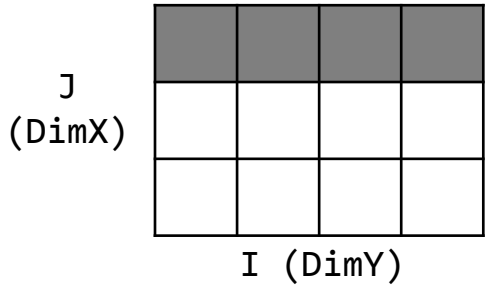
Dynamic Memory Optimization

- Nested patterns may require dynamic allocations per thread

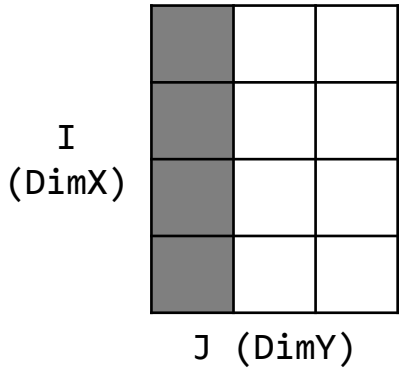
```
collection map { i =>                          // size I
   res = map { j => / * some func */ }         // size J
    …  // use of res
}
```

each thread allocates memory of size J

- Opt. 1: Allocate memory space for all threads before kernel launch (I*J)
- Opt. 2: Set proper offset and stride values for better memory accesses
  - Array access at logical index [j] => physical index [offset + j * stride]
  - Depends on the mapping decision from the analysis




```
offset = i * J
stride = 1
```




```
offset = i
stride = I
```

Code Generation

- Code generator has a set of high-level templates for each pattern
  - Just having a fixed template for each pattern is not sufficient
  - Different code structures are required for various mapping decisions
  - Generated code for sumRows example with below mapping parameters

Level 0: Dim(Y), 64, Span(1)
Level 1: Dim(X), 32, Span(all)

```
__global__ kernel(double *m, int cols, double *out) {
    int y = threadIdx.y + blockIdx.y * blockDim.y;
    __shared__ double smem[64][32]; double local_sum = 0.0;

    for (int cidx = threadIdx.x; cidx < cols; cidx += 32)
        local_sum += m[y*cols + cidx];
    smem[threadIdx.y][threadIdx.x] = local_sum;
    __syncthreads();

    /* reduce 32 values in smem[threadIdx.y][*] */

    if(threadIdx.x == 0) out[y] = smem[threadIdx.y][0];
}
```

local reduction on a registers

global reduction using shared mem

guarded instruction

Outline

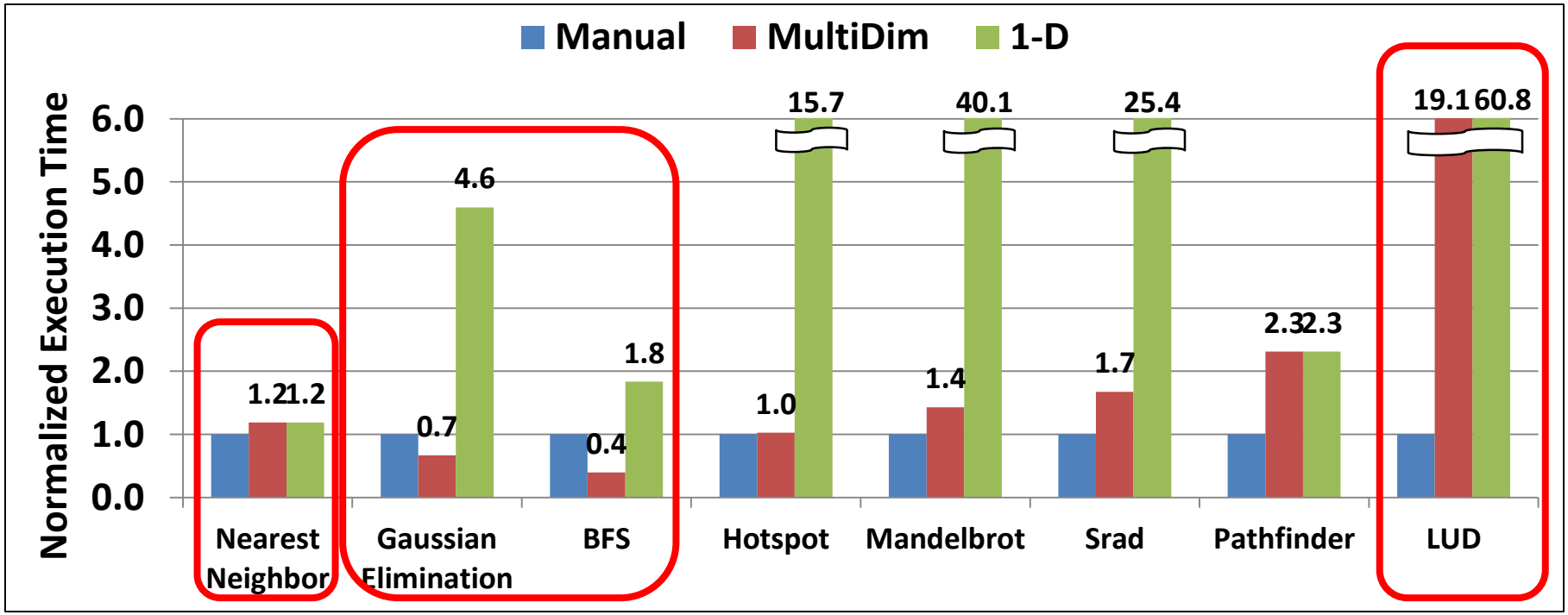
- Introduction
- Input and Output of Mapping Analysis
  - IR and Mapping Parameters
- Search for an Efficient Mapping
  - Mapping Constraints and Scores
  - Dynamic Memory Optimization
- **Evaluation**
- Conclusion

Evaluation

- Performance comparison to manually optimized CUDA
  - Applications with nested kernels in Rodinia benchmark suite
- Flexibility of our mapping analysis
  - Compare against fixed 2D strategies
- Performance impact on real-world applications
- Correlation between score and performance

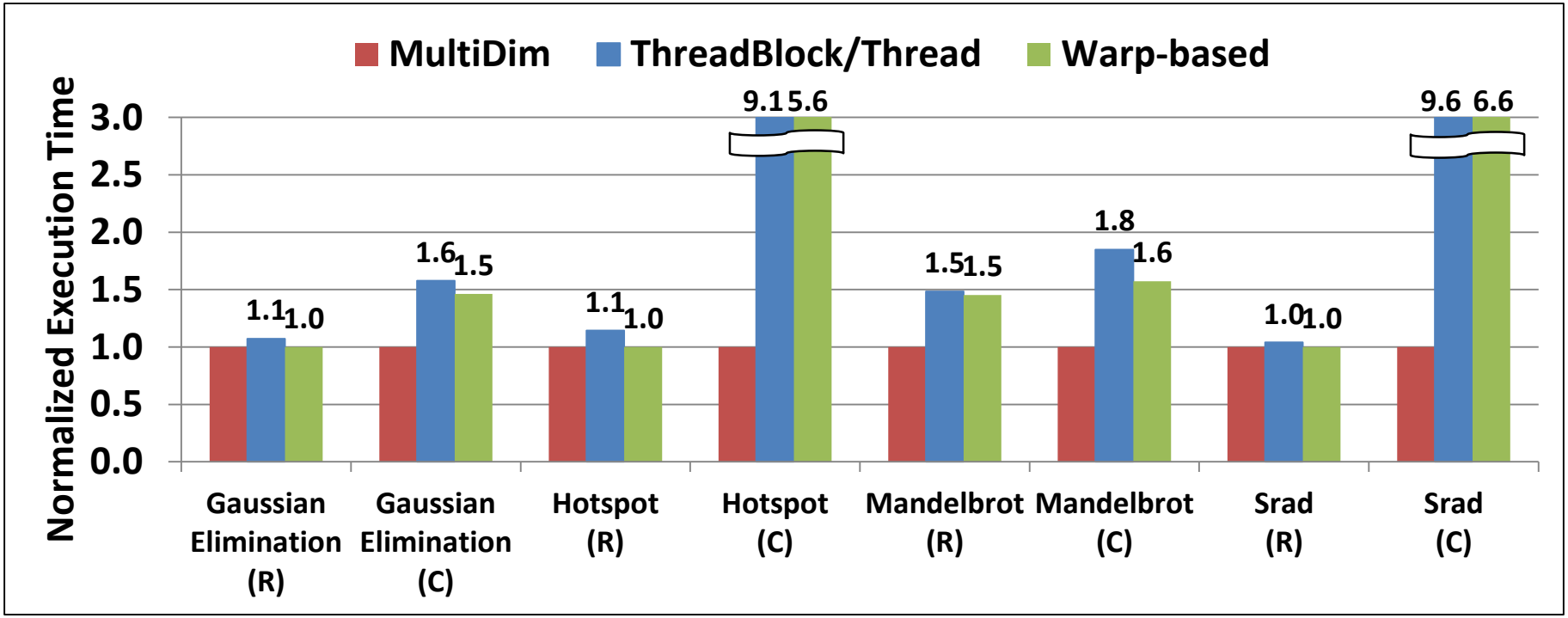
- System configuration
  - Intel Xeon X5550 (8 core, 96GB memory)
  - nVIDIA K20c GPU

Rodinia Benchmark Suite



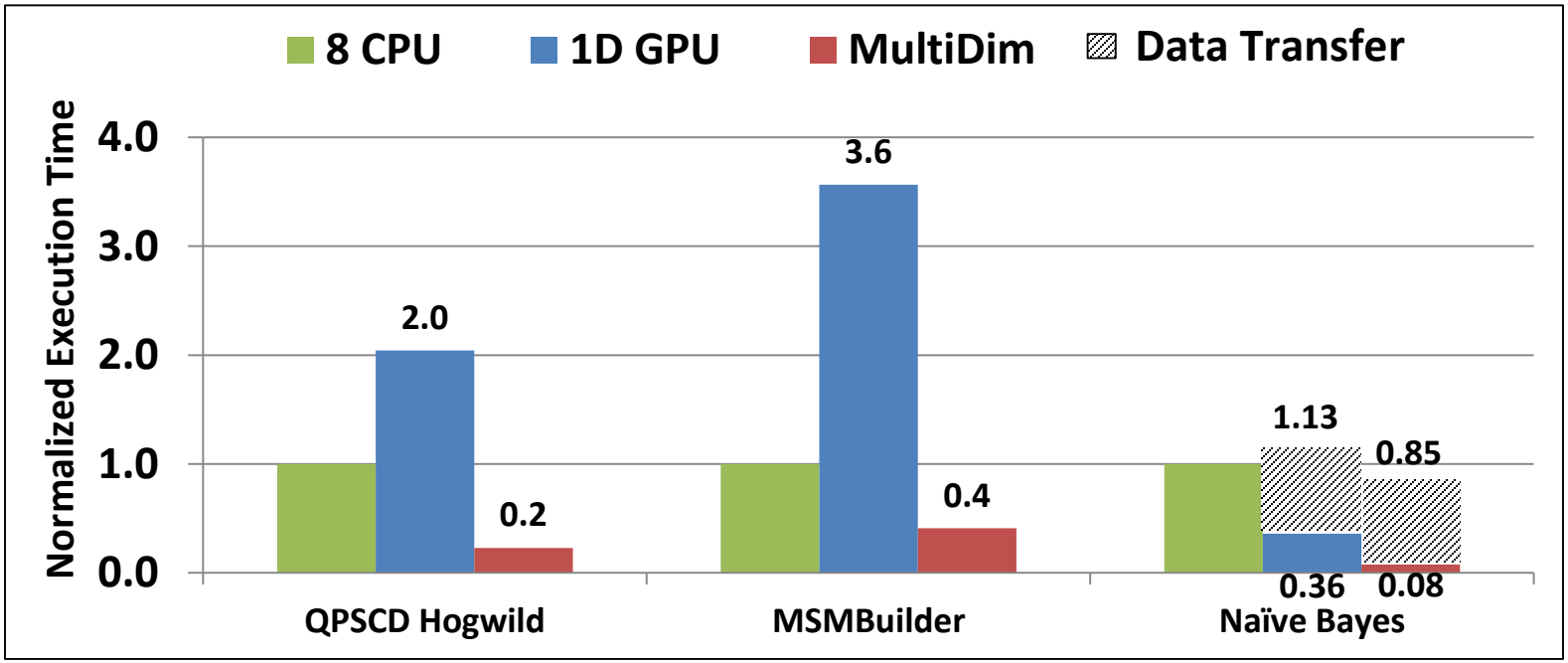
- 28.6x speedup over 1D mappings
- 24% slower than manually optimized CUDA code (7 out of 8)

Fixed 2D Mappings



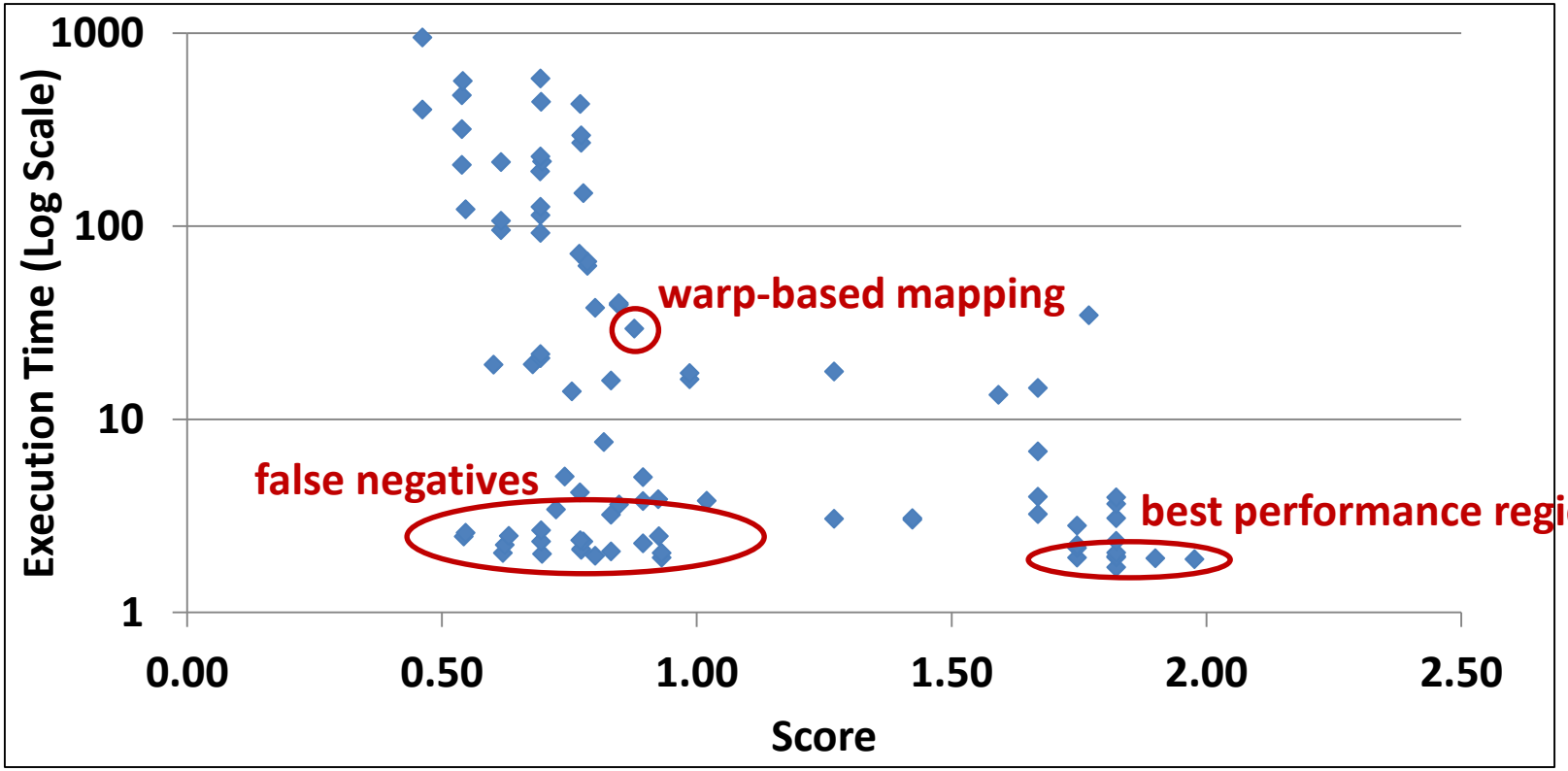
- Implemented applications in different ways (R: row-major, C: column-major)
- Up to 9.6x faster compared to fixed 2D mappings
- Our compiler is not sensitive to how the application is written

Application Case Studies



- QPSCD: quadratic programming solver with a lock-free stochastic coordinate descent
- MSMBuilder: molecular dynamics simulations and building Markov State Models
- Naïve Bayes: spam document classifier

Performance and Mapping Scores



- More detailed analytical model is required to fine tune the weights (and remove false negatives)

Conclusion

- Nested parallel patterns cannot be efficiently mapped with existing fixed mapping strategies
- We implemented a compiler analysis and optimizations to automatically find an efficient mapping based on the context
  - Define a flexible mapping parameter
  - Add mapping constraints and calculate scores
  - Add memory locality optimizations
- We demonstrated with a set of applications that our compiler automatically generate high-performance GPU code, better than manually optimized code in some cases

Thank You!

- Questions?