Locality-Aware Mapping of Nested Parallel Patterns on GPUs

HyoukJoong Lee*, Kevin Brown*, Arvind Sujeeth*, Tiark Rompf †‡, Kunle Olukotun*
*Pervasive Parallelism Lab (http://ppl.stanford.edu), Stanford University
†Purdue University, ‡Oracle Labs

Motivation

High-Level Languages for GPUs

- Provide higher productivity and portable performance
- Using parallel patterns (e.g., map, reduce, groupby) is becoming popular
  - Parallel patterns encode high-level information on parallelism and synchronization

Challenge: Parallel patterns are often nested, which are difficult to map on GPUs

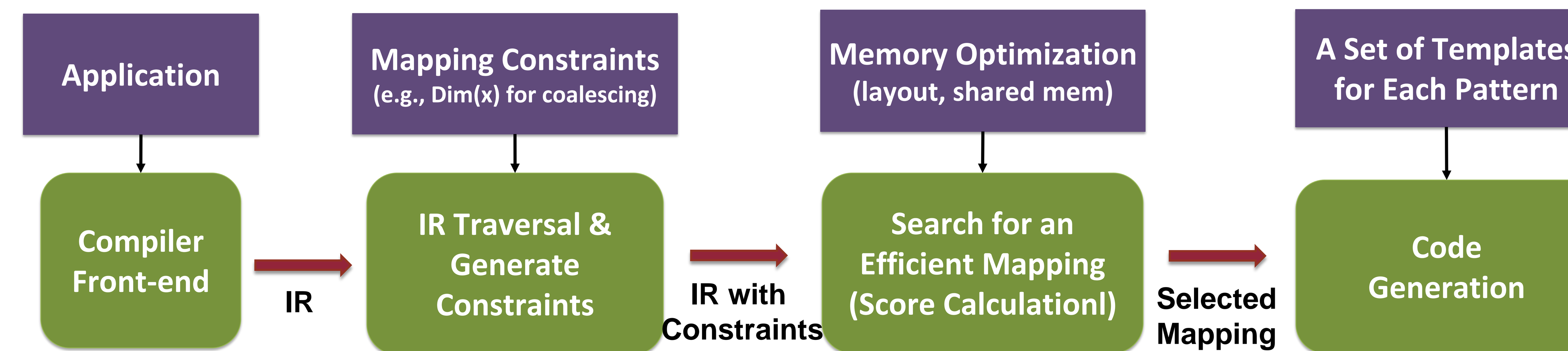
```
// Pagerank algorithm

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

- Many factors to consider together (e.g., memory coalescing, thread divergence, dynamic allocations)
- Large space of possible mappings
- Compilers typically support only a fixed mapping strategy, which is not always efficient
  - 1D mapping
  - Thread-block / thread mapping
  - Warp-based mapping

Our Contributions

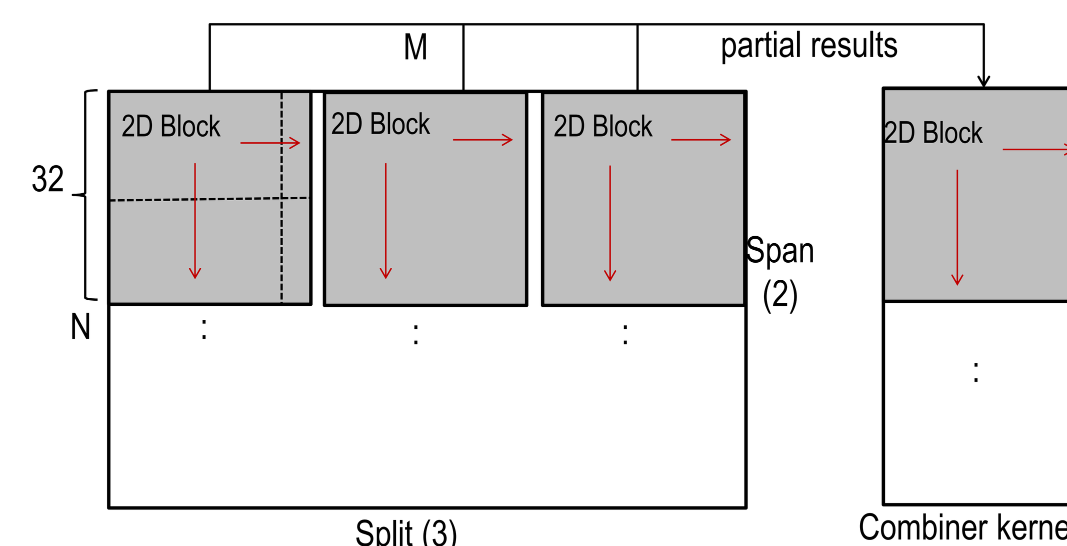
- Define mapping parameters that are general enough to cover previous mapping strategies
- Present an analysis to automatically find an efficient mapping for nested parallel patterns, maximizing locality and resource utilization
- Present compiler optimizations that interact with the mapping analysis to further improve performance, avoiding dynamic allocations and using shared memory
- Implemented a compiler and show with a set of applications that our analysis and optimizations automatically generate efficient GPU code

Compiler Flow: Analysis and Optimizations



Mapping Parameters

- **Dimension** (x, y, z, ..)
  - A logical dimension assigned to the index domain of a nest level
  - Compiler controls how indices in each dimension are mapped to hardware threads
- **Block Size** (N)
  - Number of threads assigned for a given dimension
- **Degree of Parallelism (DOP)**
  - The amount of parallel computations enabled by a mapping
  - Span(k): assign k computations to each thread on a given index domain (decreases DOP by a factor of k)
  - Span(all): assign all indices of a given index domain to the threads within a single block
  - Split(k): assign k blocks to a dimension by splitting span(all) in order to increases DOP by a factor of k, at the cost of additional kernel launch
  - Example: 2D index domain of size (N,M)
    - Split(3) on Dim x and Span(2) on Dim y, with an additional combiner kernel



- Equivalent mapping parameters for warp-based mapping

Pattern (I)	Dim(y), Size(16), Span(1)
Pattern (J)	Dim(x), Size(32), Span(all)

Mapping Constraints

- Generated while traversing the IR to prune the mapping space
- Weights are associated with each constraints

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

- Example constraints
  - For patterns that generate sequential memory requests, assign Dim(x) and block size multiple of WARP_SIZE (32)
  - For patterns that require global synchronization (e.g., Reduce), assign Span(all)

Search for an Efficient Mapping

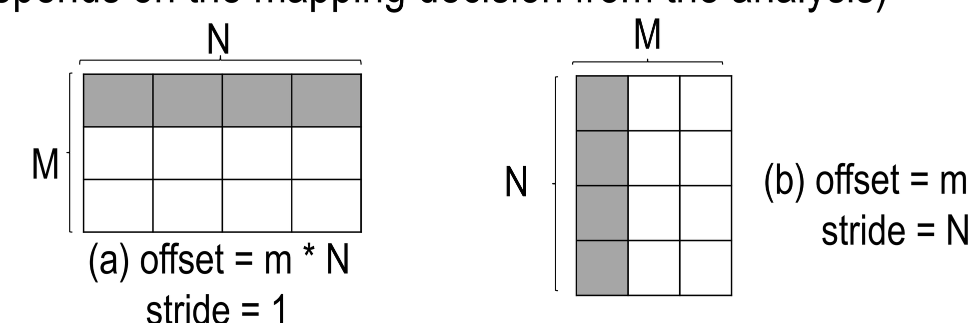
- Calculate the score of possible mappings based on constraints
  - For unknown information at compile time, assume default values (e.g., default loop size is 1000, branching factor 0.5)
- Pick one with the best score and adjust DOP
- Detailed decisions can also be adjusted at runtime
  - Changes that can be made without changing the mapping structure (e.g., thread-block size)

Dynamic Memory Allocation Optimization

- Inner patterns may require dynamic allocations

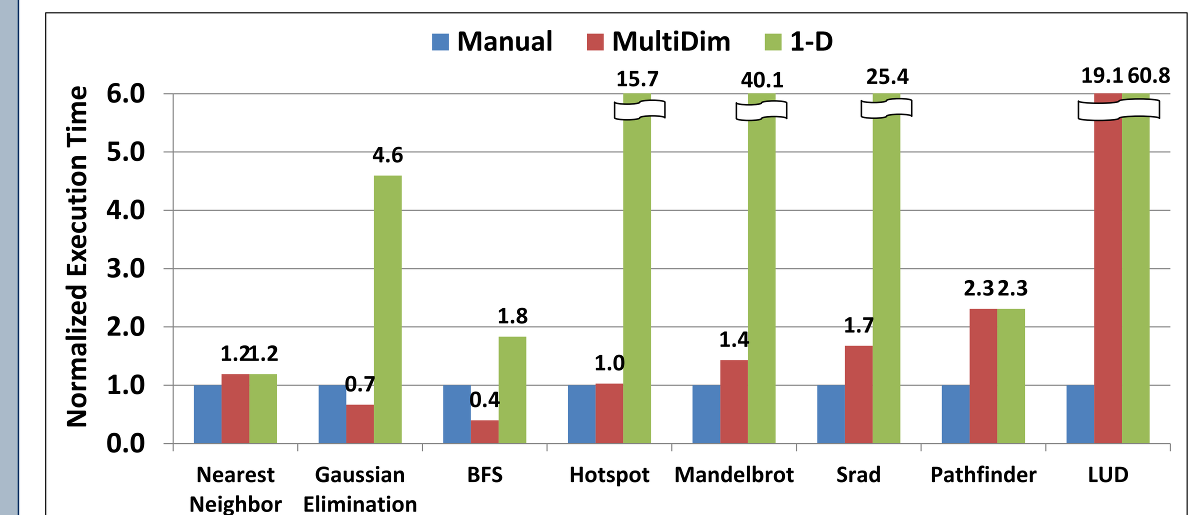
```
collection map { e =>                //size M
  // requires allocation at each e
  res = map { /* some func */ }      //size N
  // uses res
}
```

- Allocate a temporary space for the entire threads at once
- Assign a proper offset / stride values for memory coalescing (depends on the mapping decision from the analysis)



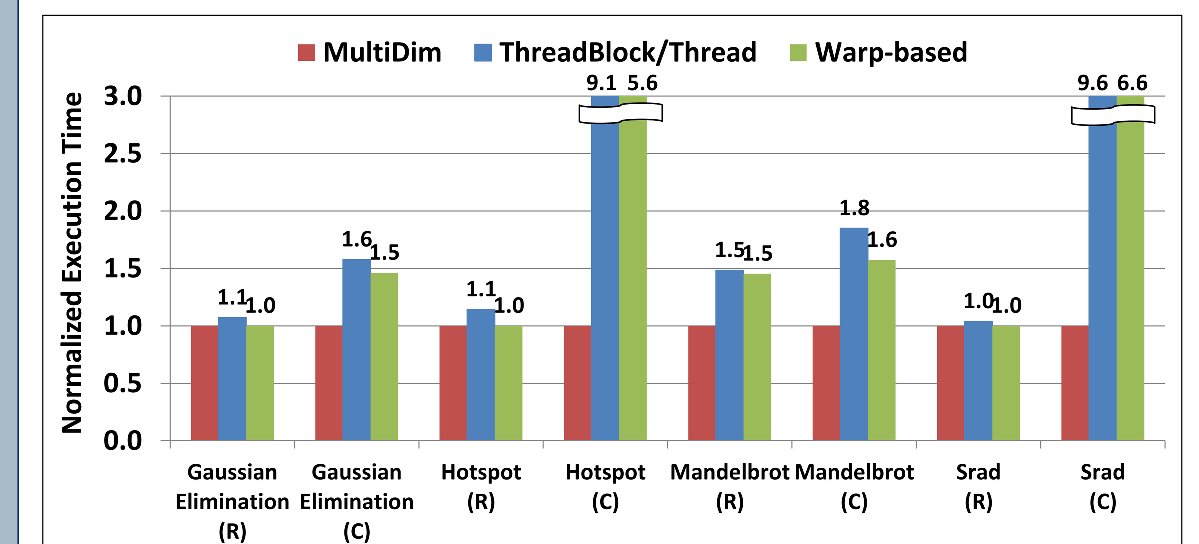
Evaluation (Nvidia K20c)

Rodinia Benchmark Suite

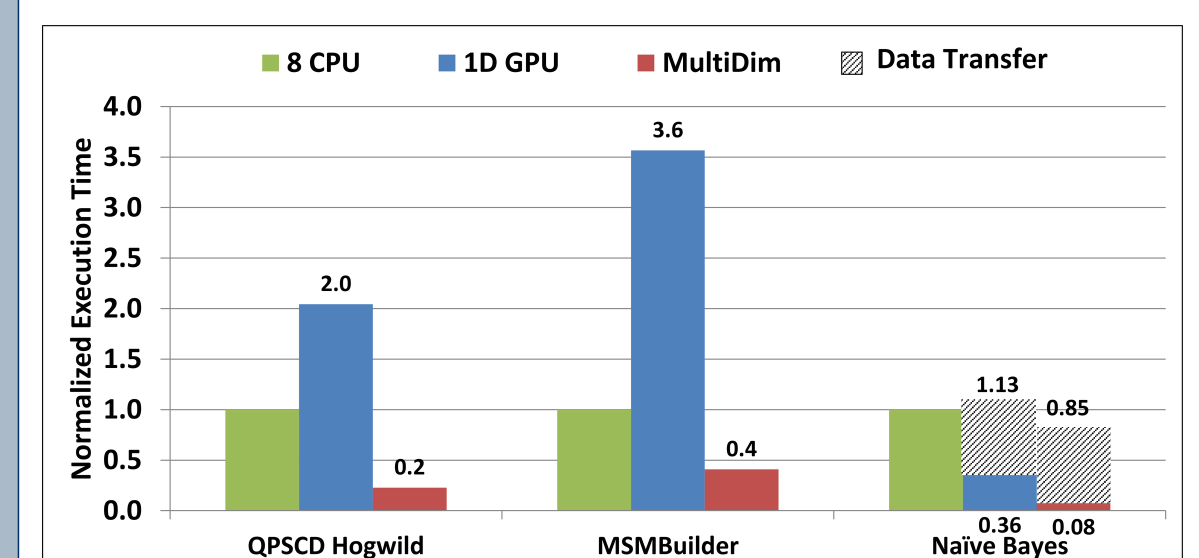


Comparison to 2D Strategies

- Applications are written in different ways (row/col major)
- Our compiler is not sensitive to how the application is written



Real World Applications



Performance vs Score

- A: best performance region, B: warp-based mapping, C: false negatives