Automated Partitioning of Data-Parallel Kernels using Polyhedral Compilation

Alexander Matz, IMC Trading B.V., Netherlands
Johannes Doerfert, Saarland University, Germany
Holger Fröning, Heidelberg University, Germany

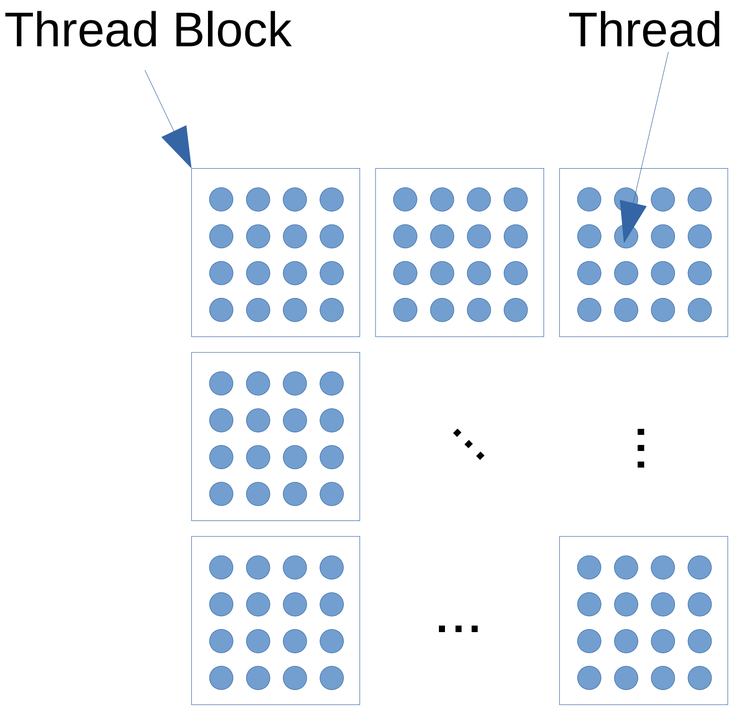
Thirteenth International Workshop on
Parallel Programming Models and Systems Software for
High-End Computing (P2S2), 2020

GPGPU Programming

Kernel Code (2D Jacobi)

```
__global__ void stencil(float* A, float* B, int N) {
  idx = threadIdx.x + blockIdx.x * blockDim.x;
  idy = threadIdx.y + blockIdx.y * blockDim.y;
  if (idx >= N || idy >= N) return;
  auto x1 = idx.x-1 >= 0 ? A[idx.y * N + idx.x-1] : 0;
  auto x2 = idx.x+1 < N ? A[idy.y * N + idx.x+1] : 0;
  auto x3 = idx.y-1 >= 0 ? A[(idx.y-1) * N + idx.x] : 0;
  auto x4 = idx.y+1 < N ? A[(idx.y+1)* N + idx.x] : 0;
  auto x5 = A[idx.y* N + idx.x];
  B[idx.y * N + idx.x] = 0.2*(x1+x2+x3+x4+x5);
}
```

Kernel Instance



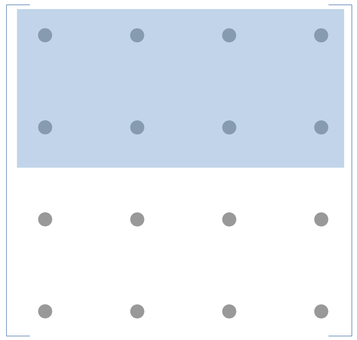
Automated Multi GPU programming?

- Thread Blocks are highly independent
- Kernels often exhibit spatial locality
- Grid provides simple mechanism to manage parallelism

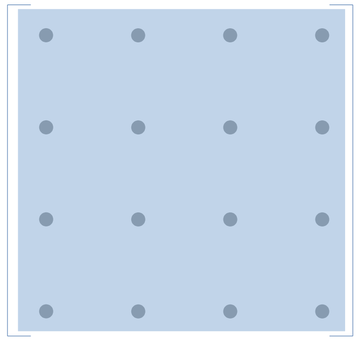
- Why not automate scale out to multiple GPUs?

GPU Data Distribution: Matrix Multiplication

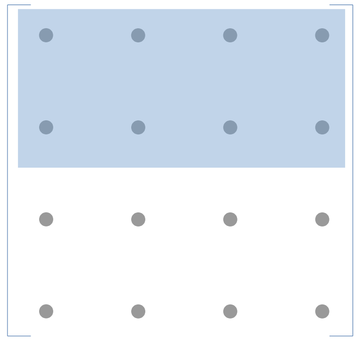
GPU 1



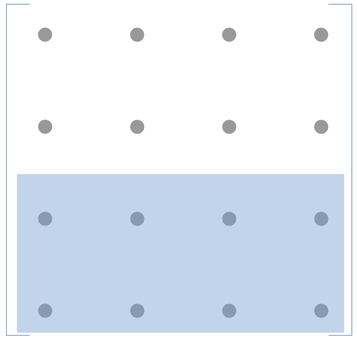
x



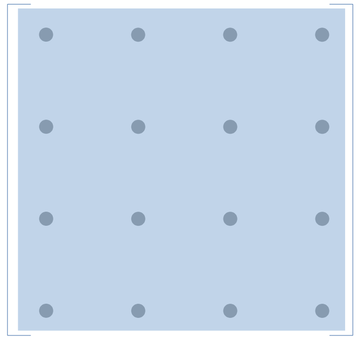
=



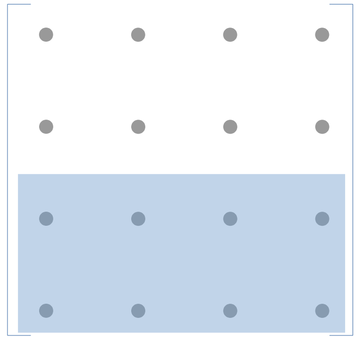
GPU 2



x



=



Polyhedral Analysis

- Mathematical model for loop optimizations
- Data dependencies are represented as sets and maps of n-dimensional integer points
- Sets described via linear constraints, maps as affine functions
- Sets and maps can be combined (union, difference, ...) and queried (min, max, ...)

```
for (int i = 0; i < n; ++i) {
    for (int j = 0; j < i; ++j) {
        S(i, j, i + j);
    }
}
```

```
D(S) := [n] -> { [i, j] : 0 <= i < n & 0 <= j < i }
   S :=       { [i, j] -> [i, j, i+j] }
```

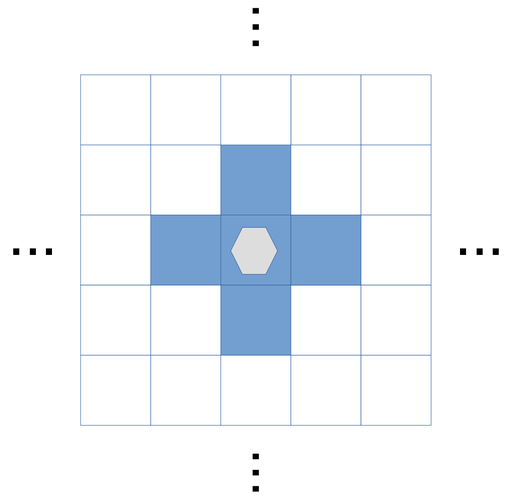
Polyhedral Analysis: 2D Jacobi

```
__global__ void stencil(float* A, float* B, int N) {
  idx = threadIdx.x + blockIdx.x * blockDim.x;
  idy = threadIdx.y + blockIdx.y * blockDim.y;
  if (idx >= N || idy >= N) return;
  auto x1 = idx.x-1 >= 0 ? A[idx.y * N + idx.x-1] : 0;
  auto x2 = idx.x+1 < N ? A[idx.y * N + idx.x+1] : 0;
  auto x3 = idx.y-1 >= 0 ? A[(idx.y-1) * N + idx.x] : 0;
  auto x4 = idx.y+1 < N ? A[(idx.y+1)* N + idx.x] : 0;
  auto x5 = A[idx.y* N + idx.x];
  B[idx.y * N + idx.x] = 0.2*(x1+x2+x3+x4+x5);
}
```

Extracted Information:

- Access patterns of a single thread
- Data type of array arguments
- Array dimensionality

Polyhedral Analysis: 2D Jacobi



```
Read(A) := [N] -> { [x, y] -> [x-1, y] : x-1 >= 0;
                    [x, y] -> [x+1, y] : x+1 < N;
                    [x, y] -> [x, y-1] : y-1 >= 0;
                    [x, y] -> [x, y+1] : y+1 < N;
                    [x, y] -> [x, y]; }
Dim(A) := [N] -> { N, inf }
ElType(A) := float

Write(B) := { [x, y] -> [x, y]; }
Dim(B) := [N] -> { N, inf }
ElType(B) := float
```

We can predict the memory accesses of all threads of a kernel!

Polyhedral Code Generation: 2D Jacobi

Goal: for a group of threads, list all array elements they read or write

1) Intersect (limit) domain of memory access map to threads satisfying:

```
[xmin, xmax, ymin, ymax] -> { [x, y] : xmin <= x < xmax and ymin <= y < ymax; }
```

The image of the map now contains all memory accesses

2) Generate efficient code that lists these elements

Polyhedral Code Generation: 2D Jacobi

For 2D Jacobi, we get a 2-level result

1) Loop that lists all rows

```
void rows(int xmin, int xmax, int ymin, int ymax, int N) {
  if (xmin >= 1) __row(xmin - 1);
  for (int c0 = xmin; c0 < xmax; c0 += 1) __row(c0);
  if (N >= xmax + 1) __row(xmax);
}
```

2) First and last element in each row

```
int first(int row, int xmin, int xmax, int ymin, int ymax) {
  return row >= xmin && xmax >= row + 2 ? ymin - 1 : xmin == row + 1 ? ymin : row - xmax + ymin
}

int last(int row, int xmin, int xmax, int ymin, int ymax) {
  return row + 1 >= xmax ? -row + xmax + ymax - 1 : xmin == row + 1 ? ymax - 1 : ymax
}
```

* Polyhedral libraries usually implement AST generation

Host Code Modifications

A custom tool (using a regex matcher) transforms code like this

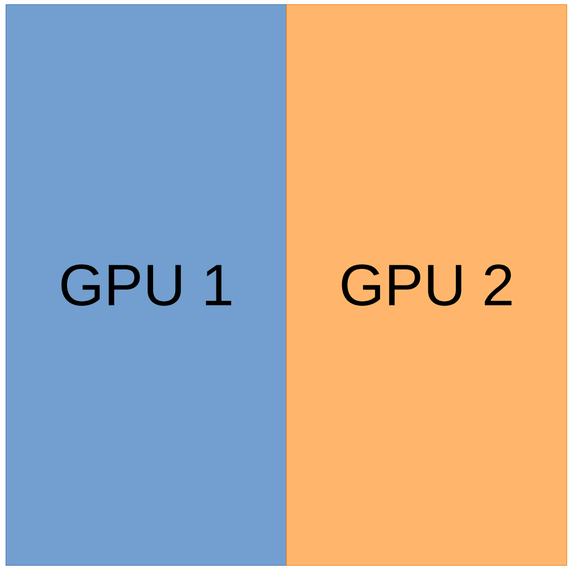
```
int main() {
  ...
  kernel<<<threadGrid, threadBlocks>>>(arr_in, arr_out, N);
  ...
}
```

Into code like this

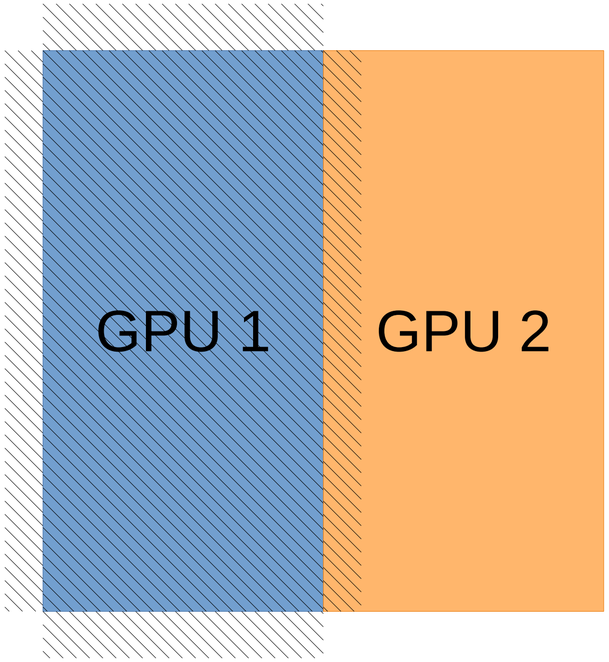
```
int main() {
  ...
  for (int 0; i < numGPUs(); ++i) {
    auto newGrid = calcGrid(threadGrid);
    redistribute_data(arr_in, newGrid);
    kernel<<<newGrid, threadBlocks>>>(arr_in, arr_out, N);
    update_distribution(arr_out, newGrid);
  }
  ...
}
```

Data redistribution

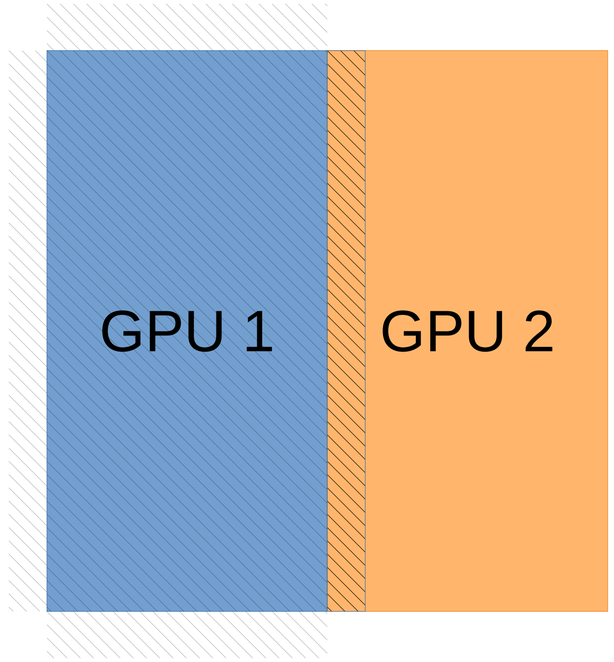
Output Matrix computed in last iteration (input in current iteration):



Overlay read pattern of input matrix (for GPU 1):



Data copied from GPU2 to GPU 1:

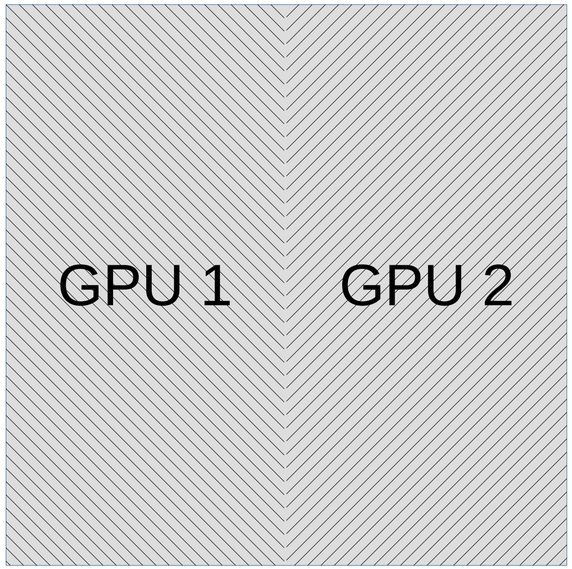


Required information for this is contained in generated code

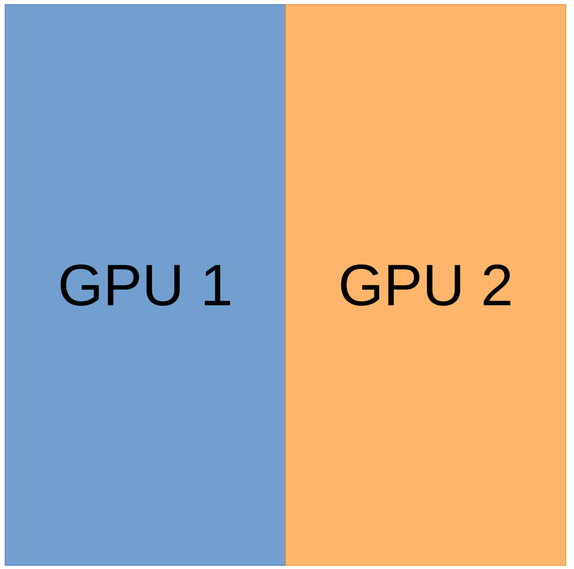
Updating data distribution

Output Matrix data distribution before first iteration:

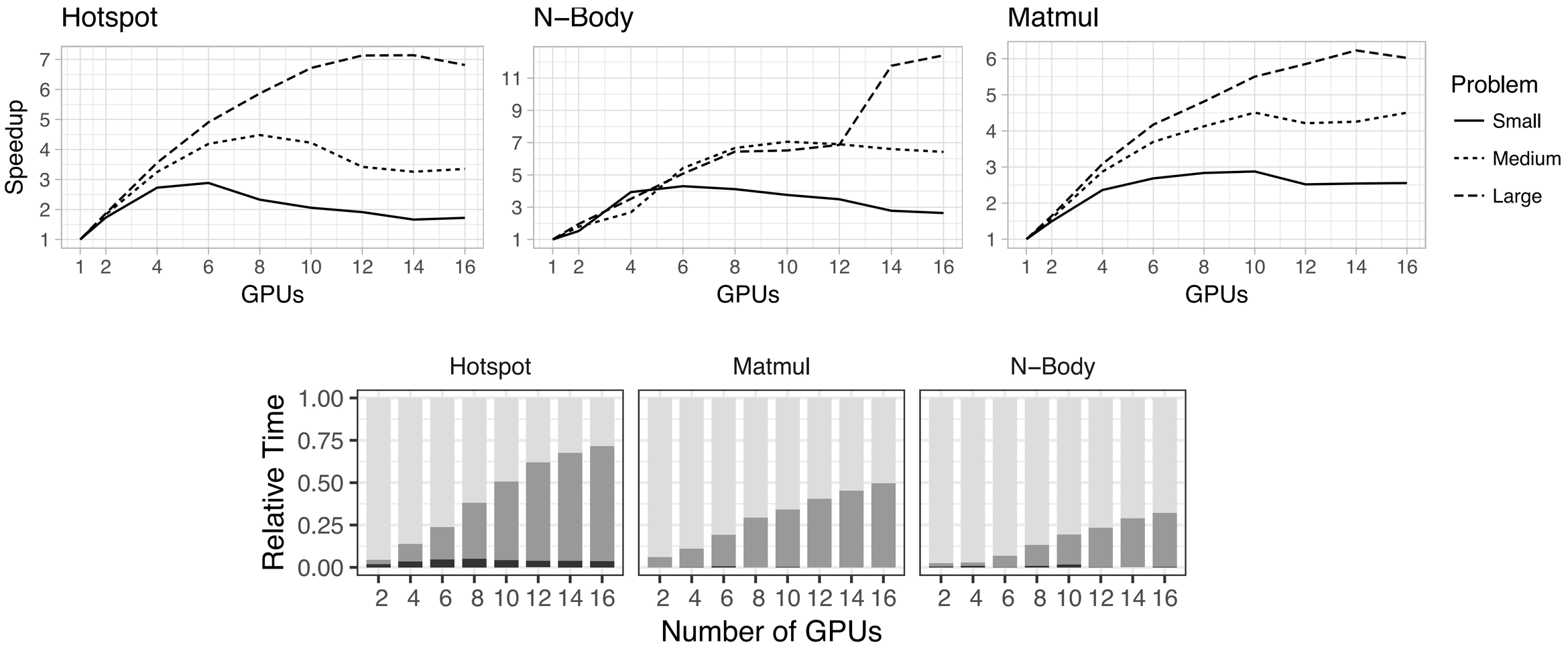
Overlay write patterns of output matrix for GPU 1 and GPU 2:



Output matrix data distribution after first iteration:



Benchmarking Results



Insights

- When polyhedral compilation works, it works well
  - Powerful and accurate model of the application
  - Limitations for irregular kernels

- Scaling out to multiple GPUs is performant for simple kernels
  - GPU applications often already optimized for locality

- Real life GPU kernels are complex
  - Robust implementation is a serious engineering effort