Towards Automation of GPU Program Optimization

I-Jui (Ray) Sung,
John Stratton, Sara S. Baghsorkhi, Christopher I. Rodrigues
Wen-Mei Hwu
The IMPACT Research Group,
University of Illinois at Urbana-Champaign

Performance Optimization for GPU Programs

- The process of manually optimizing GPU computing kernels is usually incremental
  - Some transformations enable more transformations; e.g. coarsening enable register tiling
  - Takes hours or days between steps

Performance Optimization for GPU Programs

- How can tools help?

Performance Optimization for GPU Programs

- How can tools help?
  - Automagically convert naïve GPU kernels to the optimized kernel?
    - In this case, ideally the tool's output does not have to be human-readable

Performance Optimization for GPU Programs

- How can tools help?
  - Automagically convert naïve GPU kernels to the optimized kernel?
    - In this case, ideally the tool's output doesn't have to be human-readable
    - Less likely to happen except domain-specific cases

Performance Optimization for GPU Programs

- How can tools help?
  - Automagically convert naïve GPU kernels to the optimized kernel?
    - In this case, ideally the tool's output doesn't have to be human-readable
    - Less likely to happen except domain-specific cases
  - Or, a toolbox of source code refactoring tools?

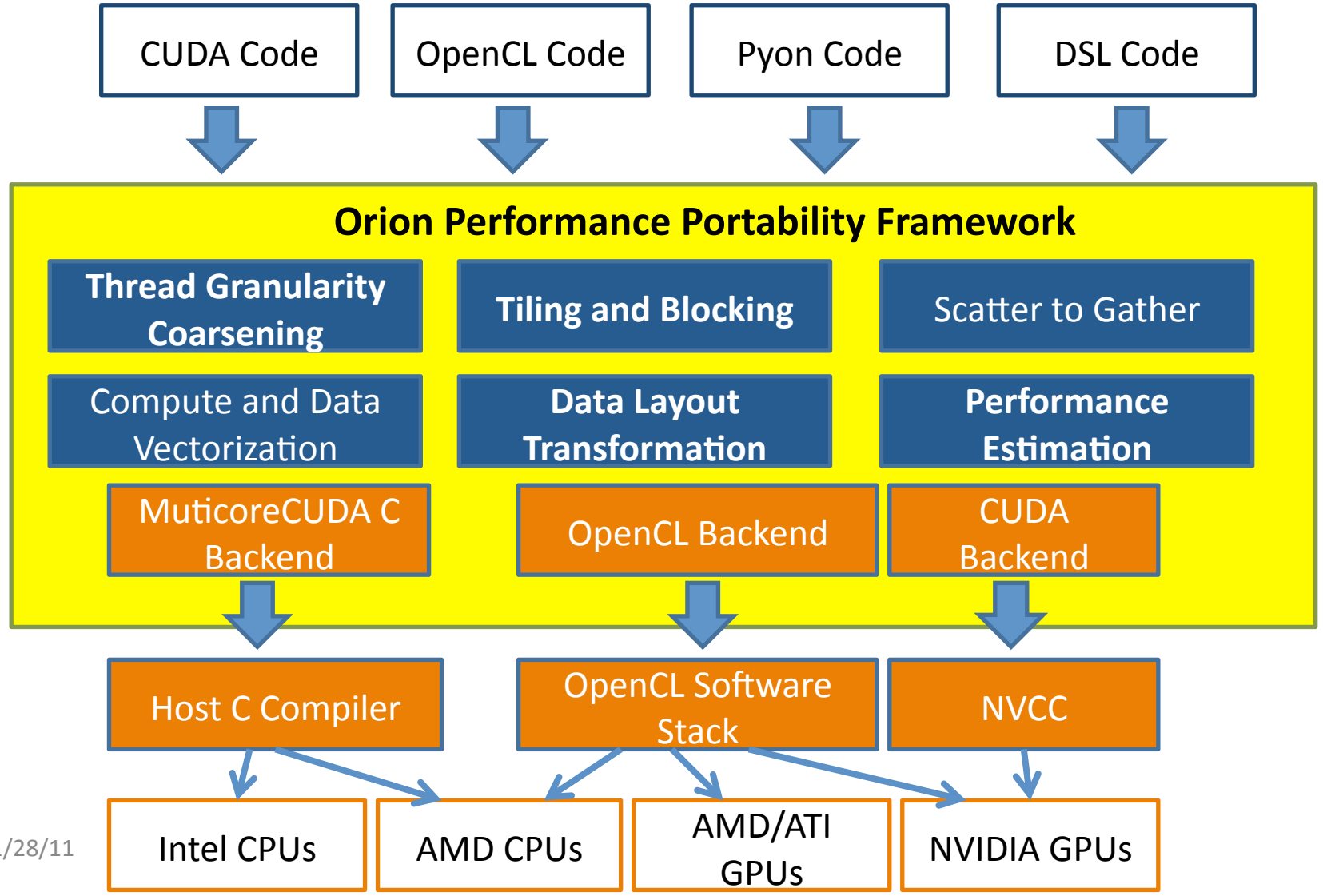
Performance Optimization for GPU Programs

- How can tools help?
  - Automagically convert naïve GPU kernels to the optimized kernel?
    - In this case, ideally the tool's output doesn't have to be human-readable
    - Less likely to happen except domain-specific cases
  - Or, a toolbox of source code refactoring tools?
    - The tool's output has to be close enough to the input, allowing the programmer to continue working on it.
      - E.g. applying domain-specific optimizations

Performance Optimization for GPU Programs

- A tool should produce minimally changed code with highlighted changes, allowing users continue working on it
  - Each change can be one or a small set of optimizations

Orion: Reducing Performance Cost of Heterogeneous Parallelism



An Example: 2D naïve Jacobi

```
#define Index2D(_nx,_i,_j) ((_i)+_nx*(_j))

__global__ void block2D_naive(float fac,
     float *A,float *Anext, int nx, int ny)
{
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  int j = blockIdx.y*blockDim.y+threadIdx.y;

  if( i>0 && j>0 &&(i<nx-1) &&(j<ny-1))
  {
    Anext[Index2D (nx, i, j)] =
      A[Index2D (nx, i, j + 1)] +
      A[Index2D (nx, i, j - 1)] +
      A[Index2D (nx, i + 1, j)] +
      A[Index2D (nx, i - 1, j)]
      - 4.0f * A[Index2D (nx, i, j)] / (fac*fac);
  }
}
```

After Coarsening Along Y

```
__global__ void block2D_naive(float fac,
     float *A,float *Anext, int nx, int ny)
{
   int i = blockIdx.x*blockDim.x+threadIdx.x;
   int j; //= blockIdx.y*blockDim.y+threadIdx.y;

   if( i>0 && (i<nx-1) )
   {
     for(j=1;j<ny-1;j++)
     {
       Anext[Index2D (nx, i, j)] =
         A[Index2D (nx, i, j + 1)] +
         A[Index2D (nx, i, j - 1)] +
         A[Index2D (nx, i + 1, j)] +
         A[Index2D (nx, i - 1, j)]
         - 4.0f * A[Index2D (nx, i, j)] / (fac*fac);
     }
   }
}
```

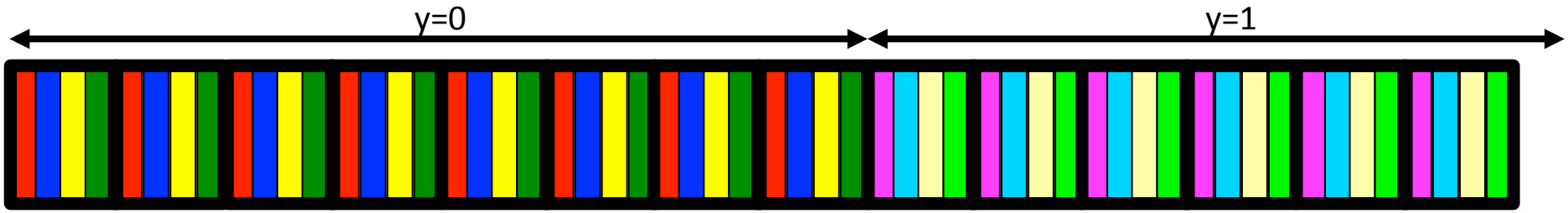
Register Tiling Exploiting Data Reuse

```
__global__ void block2D_naive(float fac,
    float *A,float *Anext, int nx, int ny)
{
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  int j; // = blockIdx.y*blockDim.y+threadIdx.y;
  float top, bottom, current;
  if( i>0 &&(i<nx-1) )
  {
    bottom = A[Index3D (nx, i, 0)];
    current = A[Index3D (nx, i, 1)];
    top = A[Index3D (nx, i, 2)];
    for(j=1;j<ny-1;j++)
    {
      Anext[Index2D (nx, ny, i, j)] =
        top +
        bottom +
        A0[Index2D (nx, i + 1, j)] +
        A0[Index2D (nx, i - 1, j)]
        - 4.0f * current / (fac*fac);
      bottom = current;
      current = top;
      top = A[Index2D (nx, ny, i, j+2)];
    }
  }
}
```

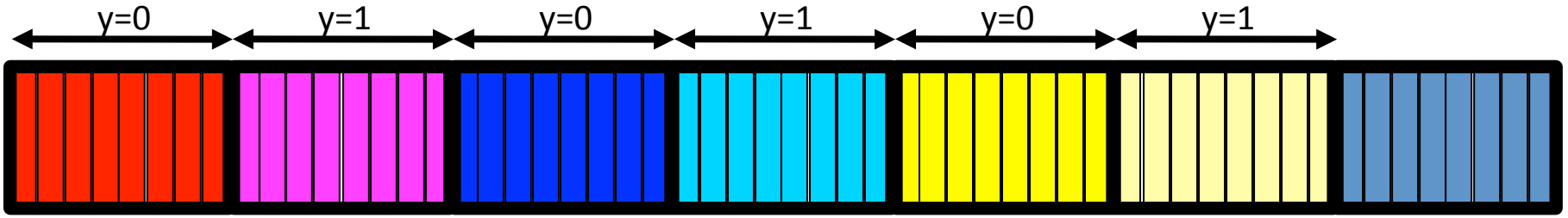
Memory Layout Transformation Tool

- Memory layout transformation is a useful transformation for GPU code
  - Array-to-Structure to Structure-of-Array transformation helps vectorization on CPU, and GPU memory coalescing on GPU

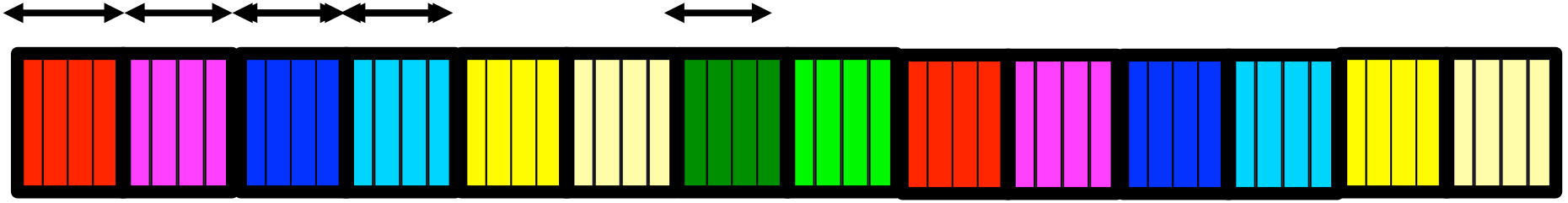
Data Layout Transformation



Array of Structure: [z][y][x][e]

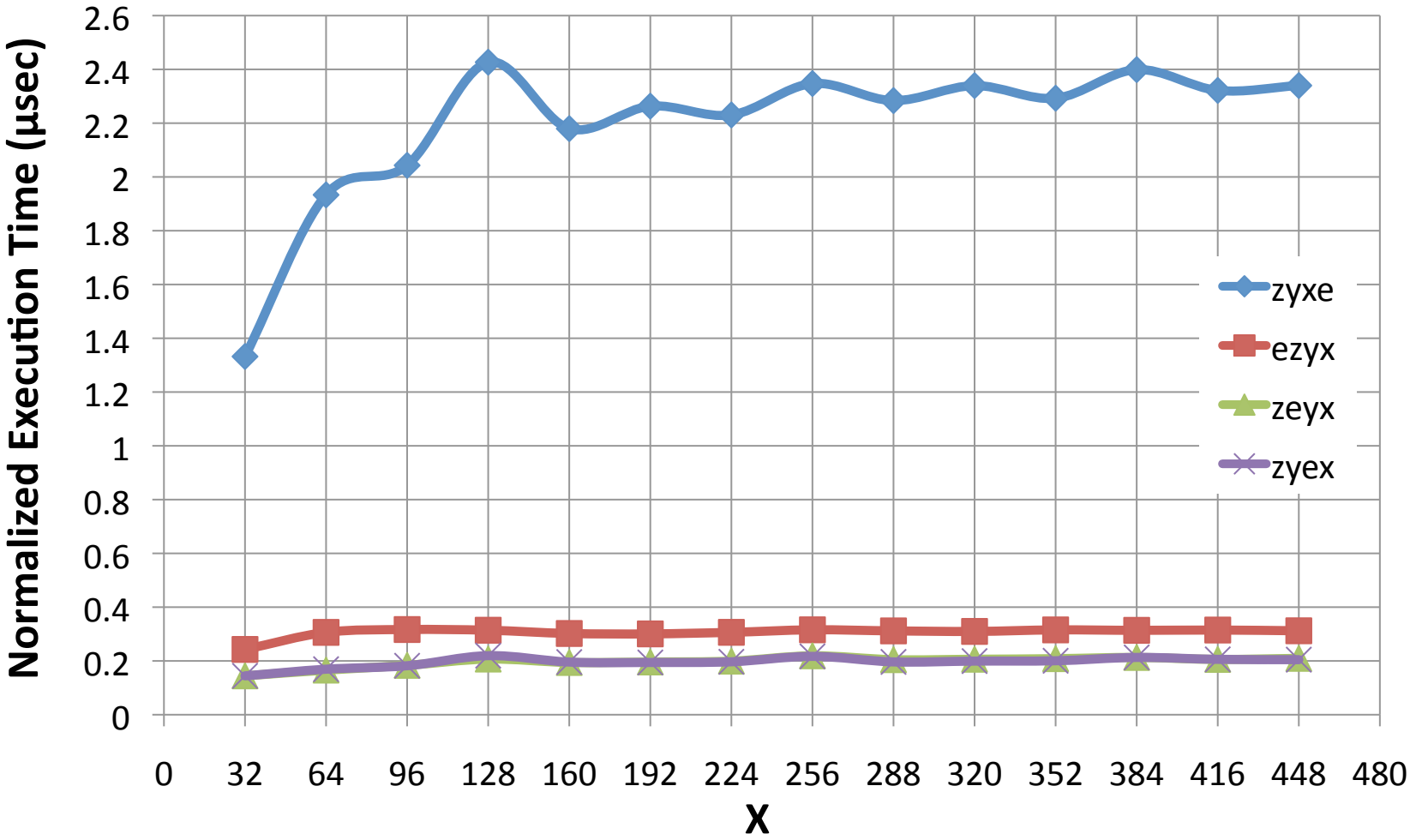


Structure of Array: [e][z][y][x]
4X faster than AoS on GTX280



$[z][y_{31:4}][x_{31:4}][e]$**$[y_{3:0}][x_{3:0}]$**, 6.6 X faster than AoS

Performance of LBM in Different Memory Layouts and Array Sizes



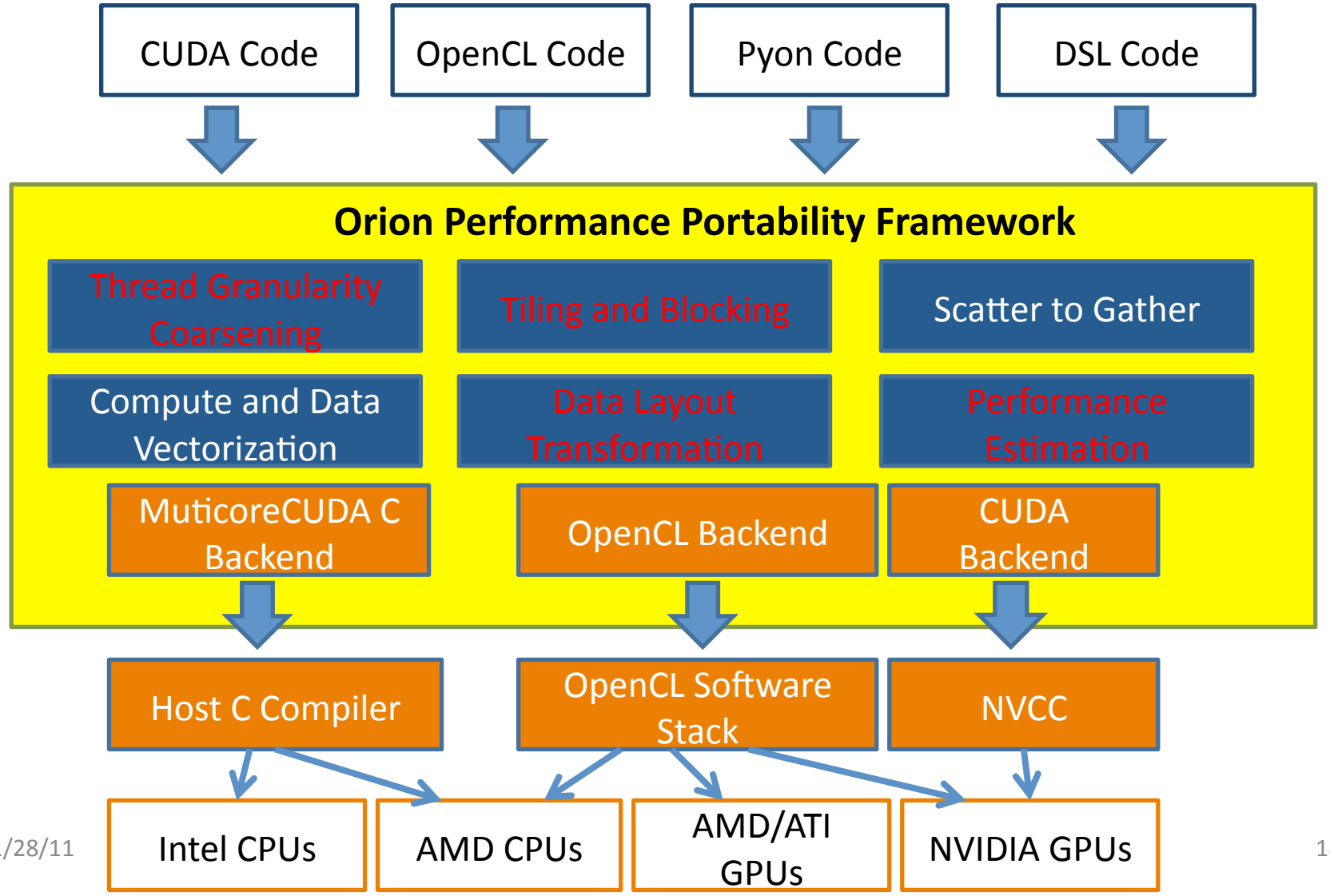
Automating Memory Layout Transformation

- Applying layout transformation manually?
  - Unlike coarsening and register tiling, data transformation, layout transformation is a global transformation
  - Marshalling code has to be introduced to guarantee correctness
- Objective: turn layout transformation into a series of incremental, local, automated changes of the application
  - Consists of a static GPU kernel refactoring tool and a run-time marshalling library
  - Run-time library helps handling data marshalling dynamically data layout transformation

ADAPT: Performance Analysis Tool for GPU Programs

- Point out the estimated cost of each statement in the CUDA kernel
  - Through static analysis and instrumentation
  - Provide concrete information like:
    - The degree of branch divergence of a given if statement
    - Extra execution cycles due to memory accesses by a statement

Orion: Reducing Performance Cost of Heterogeneous Parallelism



Conclusion

- Refactoring compiler/tools can help heavy-lifting in the process of incremental GPU program optimization
- However, compilers/tools are fragile
  - Compilers transformations need to be part of the development, rather than afterthought