Mint: An OpenMP to CUDA Translator

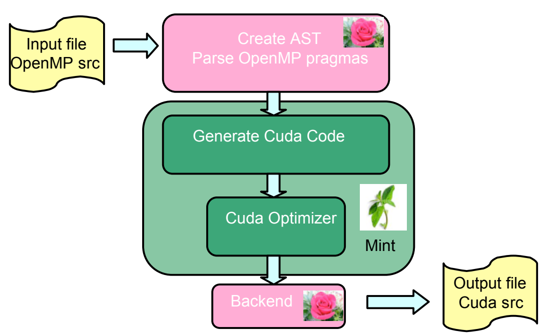
Didem Unat

UCSanDiego *Computer Science and Engineering* [simula . research laboratory]

Motivation

- OpenMP
  - Mainstream shared memory programming model
  - Few pragmas are sufficient to expresses parallelism
  - Legacy code
- GPU programming with CUDA
  - Pros
    - Provides high performance in data parallel applications
    - GPUs are cost efficient, supercomputer in a laptop
  - Cons
    - Requires tuning for the best performance
    - Computational scientists are new to memory hierarchy and data management
- Goal
  - Generate a high quality CUDA code
  - Domain specific optimizations
    - 2D and 3D structured grid problems
  - Minimal modifications in the original OpenMP source code

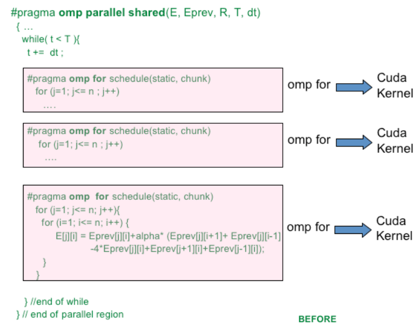
Compilation Phases of Mint



- Mint
  - Interprets and maps OpenMP pragmas into CUDA programming model
  - Performs compile-time optimizations
  - Drives the device variables for data transfer and issues necessary data copy operations
- ROSE
  - Developed at Lawrance Livermore National Lab.
  - An open source compiler framework to build source-to-source compiler
  - Mint uses ROSE to parse, translate and unparse OpenMP code
- Domain specific optimizations
  - Structured grid problems (finite difference problems)
  - Shared memory, register and kernel merge optimizations
  - Boundary condition optimizations
  - Motivating applications include heart simulations, earthquake simulations and geoscience applications

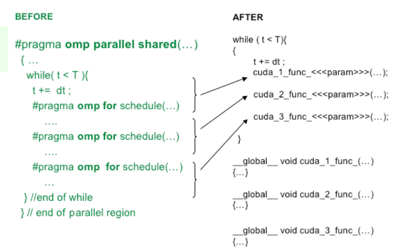
Code Transformation Steps

Step 1: Find OpenMP parallel regions and omp for loops



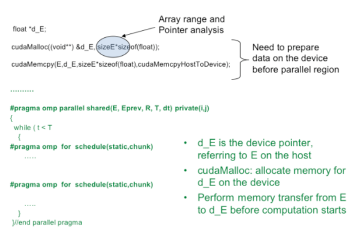
- Find omp parallel regions, these are candidates for acceleration on the GPU
- Each omp parallel for in a parallel region becomes a CUDA kernel
- Parallel regions are data regions where data should be transferred before and after these regions.

Step 2: Replace each omp for loop with a kernel launch



- Omp for loop body becomes the kernel body
- Replace the loop with a kernel launch
- Ignore the scheduling parameters within OpenMP

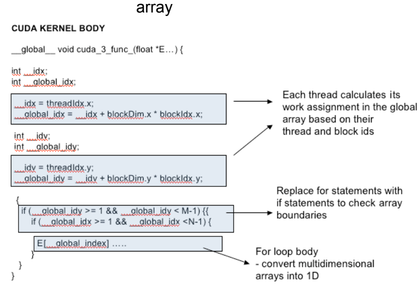
Step 3: Identify necessary data transfers



- Find the function arguments both scalar and vector variables
- Vector variables have to be allocated and transferred to the device
- Perform array range analysis and pointer analysis to find the size of an array

Step 4: Modify Kernel Body

- Replace for statements with if statements to check the array boundaries
- Calculate the assignment of a thread
- Single loop becomes 1D thread block
- Nested loops become 2D, 3D thread blocks



Optimization Steps for Finite Difference Apps

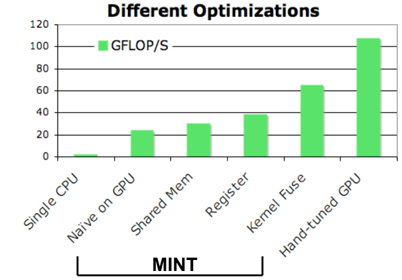
Mint performs domain specific optimizations on finite difference applications

- **Shared Memory Optimizations:** Finite difference kernels uses stencils (nearest neighbors) to compute PDEs. Their data are good candidates to place in shared memory.
  - isStencilLoop() : checks if the loop performs stencil computation. If it does, it returns the grid/array to be placed in share memory.
- **Register Optimizations:** Frequently accessed variables can be placed in registers to reduce the cost global memory accesses.
  - Mint counts the number of references to an array and finds a list of candidate variables to store in registers.
- **Kernel Fuse (for Boundary Conditions):** Boundary conditions may be in a separate loop in OpenMP implementation. However, under CUDA they can be merged into in a single kernel with a block synchronization. This reduces kernel launch and global memory access cost.

Preliminary Results

- Results for a heart simulation containing 1 PDE and 2 ODEs on Tesla C1060.
- Uses mirror boundary conditions
- Kernel-fuse optimization merges boundary condition loops and the main computation loop into a single kernel

Performance Comparison of Different Optimizations



ACKNOWLEDGMENTS

Didem Unat was supported by
a Center of Excellence grant from the Norwegian Research Council to the Center for Biomedical Computing at the Simula Research Laboratory.}