

Mint: An OpenMP to CUDA Translator

Didem Unat

Step



UCSanDiego Computer Science and Engineering [simula.research laboratory]

Code Transformation Steps

Motivation

OpenMP

- · Mainstream shared memory programming model
- Few pragmas are sufficient to expresses parallelism
- Legacy code
- · GPU programming with CUDA

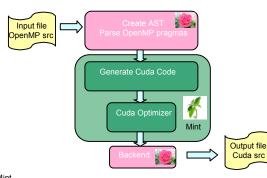
Pros

- · 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

Step 1: Find OpenMP parallel regions and omp for loops				
#pragma omp parallel shared(E, Eprev, R, T, dt)				
{ while (t < T){ t += dt;		thes	nd omp parallel regions, e are candidates for eleration on the GPU	
<pre>#pragma omp for schedule(static, chunk) for (j=1; j<= n ; j++)</pre>		ernel	ach omp parallel for in a	
<pre>#pragma omp for schedule(static, chunk) for (j=1; j<= n; j++)</pre>		ernel CUE	allel region becomes a DA kernel	
<pre>fpragma omp for schedule(static, chunk) for (j=1; j=c n; j+>){ for (j=1; i=c n; i+>){ for (j=1; i=c n; i+>){ Eli][0 = Eprev[]][4a]pha* (Eprev[][1]+1]+ Eprev[]][Eli][0 = Eprev[][1]+Eprev[]+1][0]+Eprev[]+Eprev[]+1][0]+Eprev[]+Eprev[]+1][0]+Eprev[]+1][0]+Eprev[]+1][0]+Eprev[]+</pre>		regio _{uda} be ti	arallel regions are data ons where data should ransferred before and r these regions.	
} //end of while } // end of parallel region	BEFORE		AFTER	
Step 2: Replace each omp for	#pragma omp p	arallel shared(while (t < T){	
• Omp for loop body becomes the kernel body	{ while(t < T){ t+= dt;	for schedule()	<pre>{ t+= dt; cuda_1_func_<<<pre>cuda_2_func_<<<pre>cuda_2_func_<<<pre>cuda_3_func_<<<pre>second</pre></pre></pre></pre></pre>	
Replace the loop with a kernel launch		for schedule()	} global void cuda_1_func_() {}	
Ignore the scheduling parameters within OpenMP	} //end of while } // end of parallel	I region	global void cuda_2_func_() {} global void cuda_3_func_() {}	
fort "4.E Pointer analysis Step 3: Identify necessary				
cudaMalloc((void**) &d_E, sizeE*sizeof(float)); Nee	d to prepare	0a	ata transfers	
	i on the device ire parallel region		unction arguments both vector variables	
(while (1 < T (Boragma omp for schedule(static,chunk) Brand De terring to E or cutdaMalloc: allc	 d <u>E</u> is the device pointer, referring to <u>E</u> on the host cudaMalloc: allocate memory for 		Vector variables have to be allocated and transferred to the device	
Perform memor	y transfer from E omputation starts		rray range analysis and lysis to find the size of an	
Step 4: Modify Kernel Body	CUDA KERNEL BODY alobal void cuda 3 func	(float *E) {		
Replace for statements with if statements to check the array boundaries	intidz: intobsal_idz: 			
Calculate the assignment of a thread	intdiy: Inread and block ids (itdiy=threadite.y:diy=threa			
Single loop becomes 1D thread block	{ if (idy >= 1 && if (idy >= 1 && [E[idy == 1 &&		Replace for statements with if statements to check array boundaries	
Nested loops become 2D, 3D thread blocks	} }		For loop body - convert multidimensional arrays into 1D	

Optimization Steps for Finite Difference Apps

Mint performs domain specific optimizations on finite difference applications

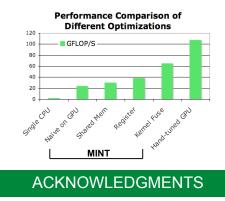
- · Shared Memory Optimizations: Finite difference kernels uses stencils (nearest neighbors) to compute PDEs. Their data are good candidates to place in shared memory.
- · isStencilLoop() : checks if the loop performs stencil computation. If it does, it returns the grid/array to be placed in share memory.
- · 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



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.}