

Extending Abstract GPU APIs to Shared Memory

Ferosh Jacob

Department of Computer Science, University of Alabama, USA

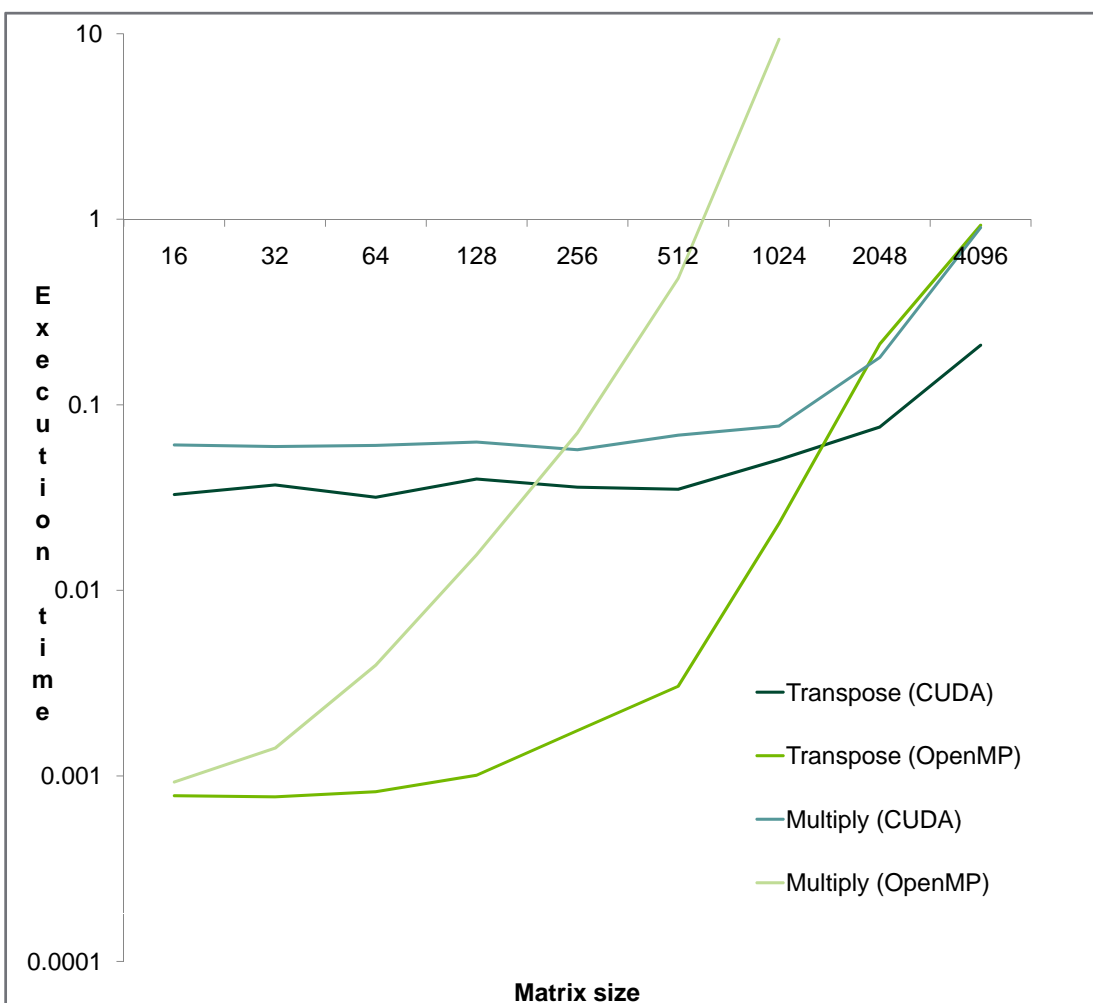
Contact: fjacob@crimson.ua.edu

Advisor: Dr. Jeff Gray

Introduction

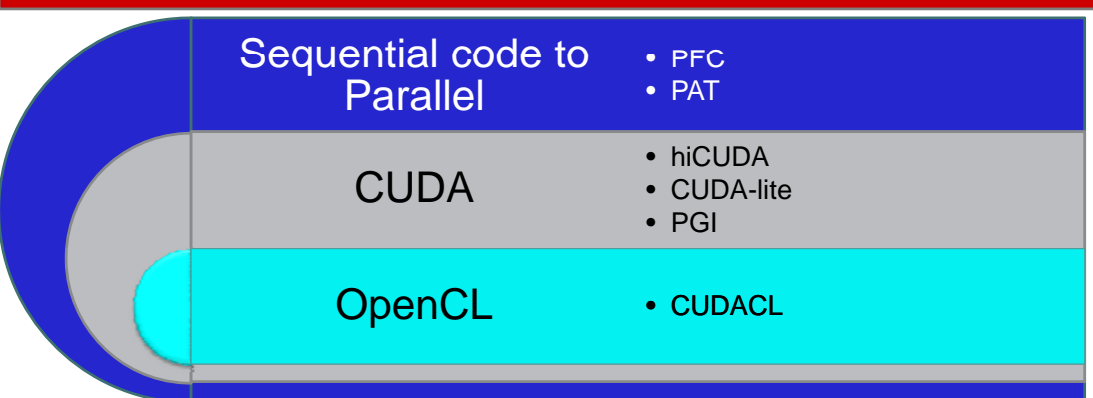
Parallel programming can be defined as the creation of code for computations that can be executed simultaneously. Due to their highly parallelized structure, Graphical Processing Units (GPUs) provide an excellent platform for executing parallel programs. NVIDIA's Computation Unified Device Architecture (CUDA), Microsoft's Direct Compute and Khronos Group's OpenCL are the most commonly used frameworks for General-Purpose GPU (GPGPU) programming. Multicore and GPU programming still requires skill beyond that of an average programmer. Currently, in order to write a program that will execute a block of code in parallel, a programmer must learn a parallel programming Application Programming Interface (API) that can be used to describe the computation. Even after the execution, the programmer must use other APIs or frameworks to evaluate the performance of their parallel program.

A comparison study of OpenMP and CUDA



A comparison study of Matrix multiplication and Matrix transpose is shown in the figure above. The GPU programs are adapted from an NVIDIA CUDA installation package to ensure high performance. As seen from the figure, as the data size increases the GPU has an advantage over the shared memory (as realized by OpenMP), but the crossing point depends on factors like complexity of the program, configuration of the machine in which program is executed, and language-specific details. This makes the decision on which architecture to choose challenging even for experts. The result is that code ends up being written for both architectures.

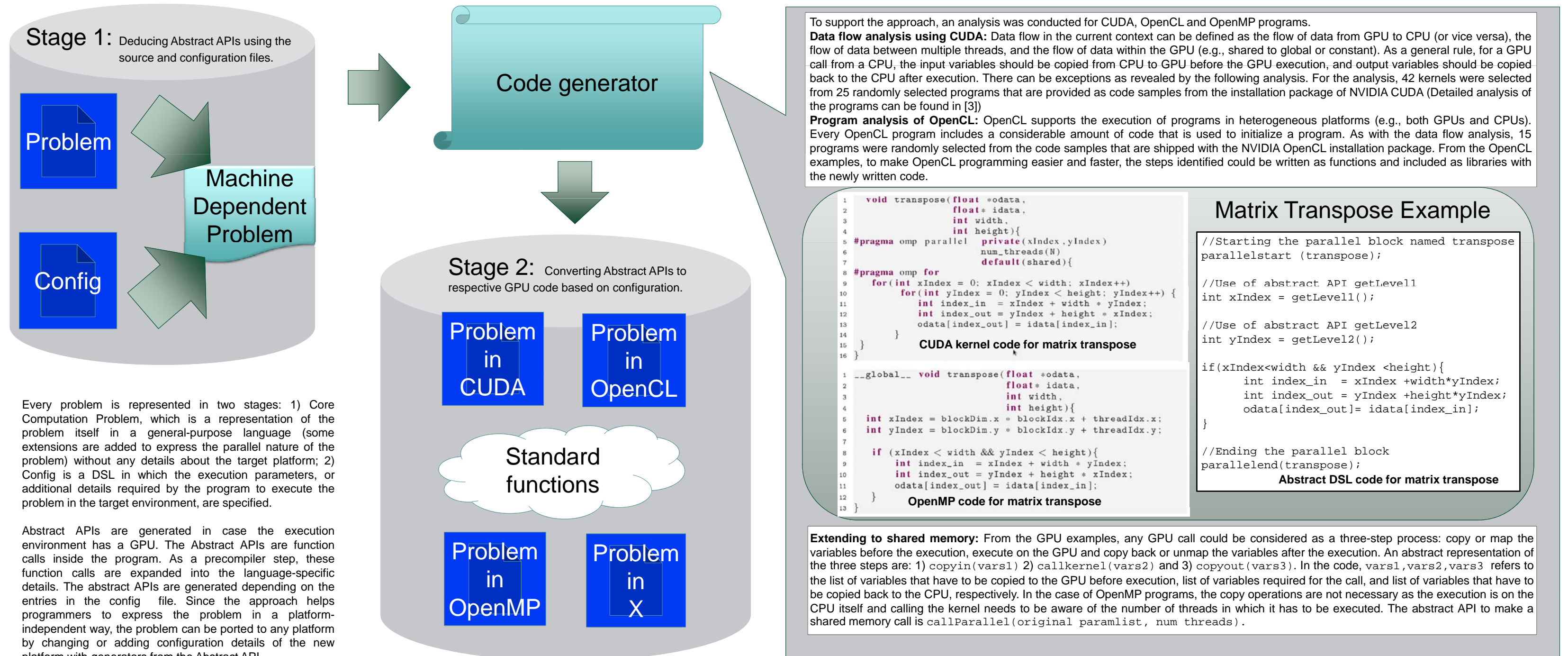
Related Work



References

1. F. Jacob, R. Arora, P. Bangalore, M. Mernik, and J. Gray, "Raising the level of abstraction of GPU-programming," in *Proceedings of the 16th International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, July 2010, pp. 339–345.
2. F. Jacob, D. Whittaker, S. Thapaliya, P. Bangalore, M. Memik and J. Gray, "CUDA CL: A tool for CUDA and OpenCL Programmers," in *Proceedings of the International Conference on High Performance Computing*, Goa, India, in press.
3. <http://cs.ua.edu/graduate/fjacob/software/analysis/>
4. T. D. Han and T. S. Abdelrahman, "hiCUDA: A high-level directive-based language for GPU programming," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, Washington, D.C., March 2009, pp. 52–61.
5. <http://www.gpugroup.com>

Internal details of the proposed approach



Configuration GPU programs using CUDACL

When the programmer wants to create a parallel block, he or she clicks the starting and ending lines in the editor (Eclipse). CUDACL responds by highlighting the line numbers and prompts for a name of the parallel block. After entering the name of the block, the tool creates a ".gpl" file. In the form, there are multiple sections corresponding to different configuration parameters. A short description of each of the sections is given below.

```

9 /*compute path for each point*/
10 for(i = 0; i<pSize; i++){
11     ComputePath(points[i], depth, lBound,
12                rBound, &path[i]);
13 }
14
    
```

Defining parallel block in the octtree program

GPU execution parameters can be expressed in terms of integer numbers or in terms of any variable available in the context. There is another option to use the OpenCL APIs to find the work group or work item size.

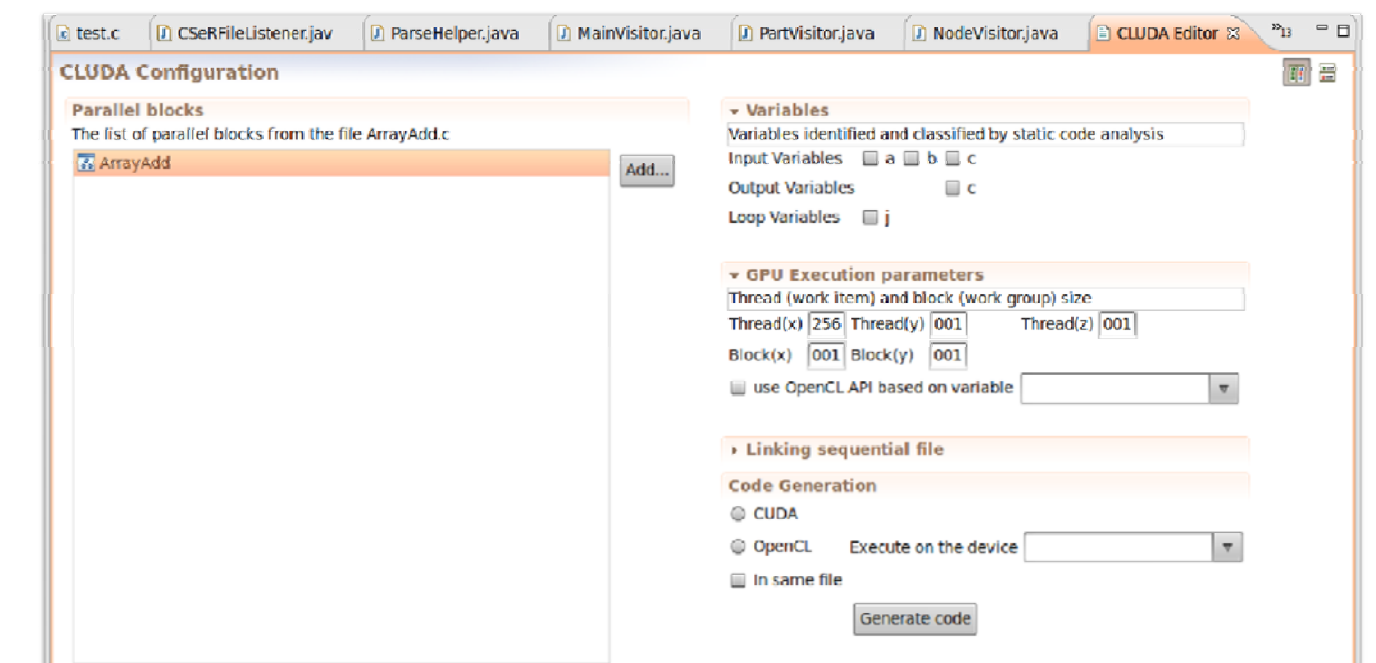
Linking sequential file. This section links the parallel block defined by the programmer with the file, the fields of the section include starting line number, ending line number and name of the parallel block. The name of the parallel block would be the same as the generated GPU method. This helps to modify the scope of the parallel block from its initial value. Scope is defined in terms of line numbers in the source code.

Code generation has options to generate CUDA or OpenCL code. For OpenCL, a specific target device can be specified. Even if the device is not specified, it finds out all of the devices and executes on the first one available. Another option in this section specifies how to separate the GPU code.

Variables. In this section, the programmer specifies the variables that should be copied into the GPU before execution of the kernel, the variables that should be copied into the CPU after the execution, and also the loop variables.

Loop variables are defined as any variable that the programmer is not interested in during the GPU computation. Initially, CUDACL shows the input, output, and loop variables based on the calculation and the programmer is free to manually configure these. The variable copy can be implemented in CUDA either using mapping or explicit copy (the current approach uses explicit copy).

Parallel blocks. Any file can have one or more parallel blocks. All the other sections are dependent on the parallel block.



Configuring Execution of ArrayAdd using CUDACL

```

for (j = 0; j < N; j++) {
    c[j] = a[j] + b[j];
}
startline 13
endline 17
name ArrayAdd

void GPU Method ArrayAdd(int[] a, int[] b, int[] c, int N) {
    int j = getGlobalId();
    if (j < N) {
        c[j] = a[j] + b[j];
    }
}
Code generated for the kernel code for ArrayAdd
    
```