Extending Abstract GPU APIs to Shared Memory

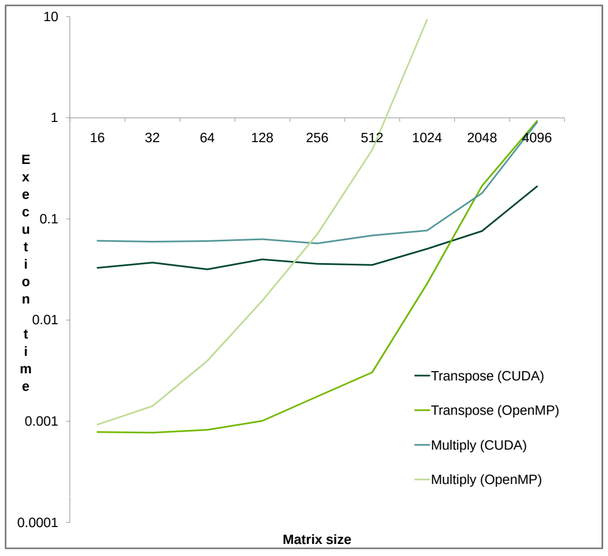
Ferosh Jacob
Department of Computer Science, University of Alabama, USA
Contact: *fjacob@crimson.ua.edu*
Advisor: Dr. Jeff Gray

THE UNIVERSITY OF ALABAMA ENGINEERING

Introduction

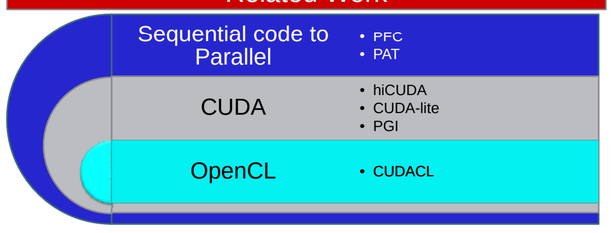
Parallel programming can be defined as the creation of code for computations that can be executed simultaneously. Due to their highly parallelized structure, Graphical Processing Units (GPUs) provide an excellent platform for executing parallel programs. NVIDIA's Computation Unified Device Architecture (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.

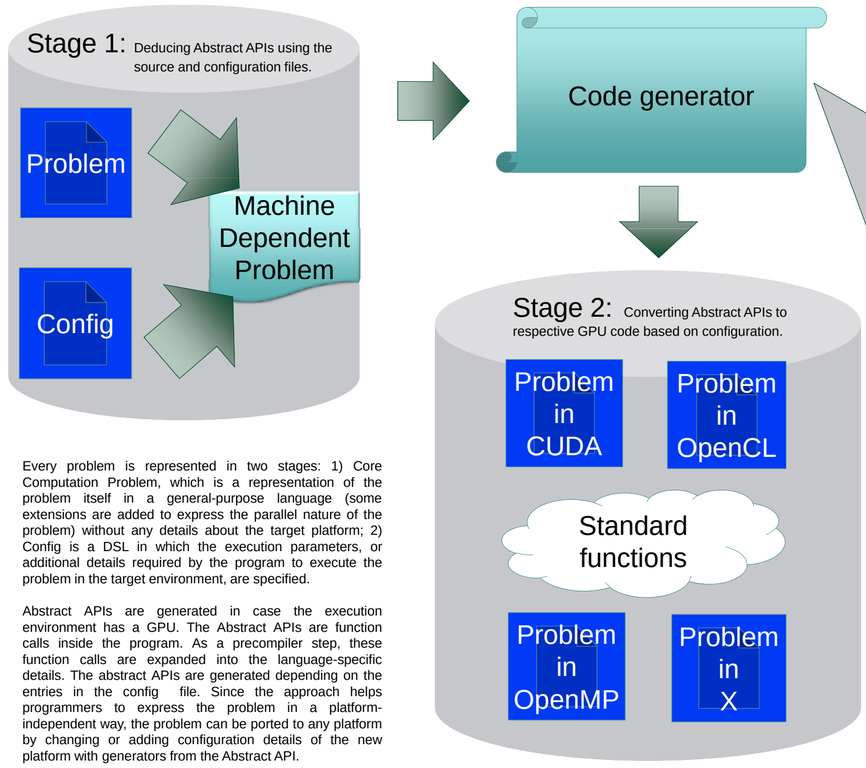
Related Work



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. Mernik and J. Gray, "CUDACL: 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.pgroup.com*

Internal details of the proposed approach



Every problem is represented in two stages: 1) Core Computation Problem, which is a representation of the problem itself in a general-purpose language (some extensions are added to express the parallel nature of the problem) without any details about the target platform; 2) Config is a DSL in which the execution parameters, or additional details required by the program to execute the problem in the target environment, are specified.

Abstract APIs are generated in case the execution environment has a GPU. The Abstract APIs are function calls inside the program. As a precompiler step, these function calls are expanded into the language-specific details. The abstract APIs are generated depending on the entries in the config file. Since the approach helps programmers to express the problem in a platform-independent way, the problem can be ported to any platform by changing or adding configuration details of the new platform with generators from the Abstract API.

To support the approach, an analysis was conducted for CUDA, OpenCL and OpenMP programs.
Data flow analysis using CUDA: Data flow in the current context can be defined as the flow of data from GPU to CPU (or vice versa), the flow of data between multiple threads, and the flow of data within the GPU (e.g., shared to global or constant). As a general rule, for a GPU call from a CPU, the input variables should be copied from CPU to GPU before the GPU execution, and output variables should be copied back to the CPU after execution. There can be exceptions as revealed by the following analysis. For the analysis, 42 kernels were selected from 25 randomly selected programs that are provided as code samples from the installation package of NVIDIA CUDA (Detailed analysis of the programs can be found in [3])
Program analysis of OpenCL: OpenCL supports the execution of programs in heterogeneous platforms (e.g., both GPUs and CPUs). Every OpenCL program includes a considerable amount of code that is used to initialize a program. As with the data flow analysis, 15 programs were randomly selected from the code samples that are shipped with the NVIDIA OpenCL installation package. From the OpenCL examples, to make OpenCL programming easier and faster, the steps identified could be written as functions and included as libraries with the newly written code.

```
void transpose(float *odata,
               float* idata,
               int width,
               int height){
#pragma omp parallel   private(xIndex,yIndex)
                       num_threads(N)
                       default(shared){
#pragma omp for
   for(int xIndex = 0; xIndex < width; xIndex++)
        for(int yIndex = 0; yIndex < height; yIndex++) {
            int index_in  = xIndex + width * yIndex;
            int index_out = yIndex + height * xIndex;
            odata[index_out] = idata[index_in];
        }
 }
}
```

CUDA kernel code for matrix transpose

```
__global__ void transpose(float *odata,
                          float* idata,
                          int width,
                          int height){
  int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

  if (xIndex < width && yIndex < height){
      int index_in  = xIndex + width * yIndex;
      int index_out = yIndex + height * xIndex;
      odata[index_out] = idata[index_in];
  }
}
```

OpenMP code for matrix transpose

Matrix Transpose Example

```
//Starting the parallel block named transpose
parallelstart (transpose);

//Use of abstract API getLevel1
int xIndex = getLevel1();

//Use of abstract API getLevel2
int yIndex = getLevel2();

if(xIndex<width && yIndex <height){
      int index_in  = xIndex +width*yIndex;
      int index_out = yIndex +height*yIndex;
      odata[index_out]= idata[index_in];
}

//Ending the parallel block
parallelend(transpose);
```

Abstract DSL code for matrix transpose

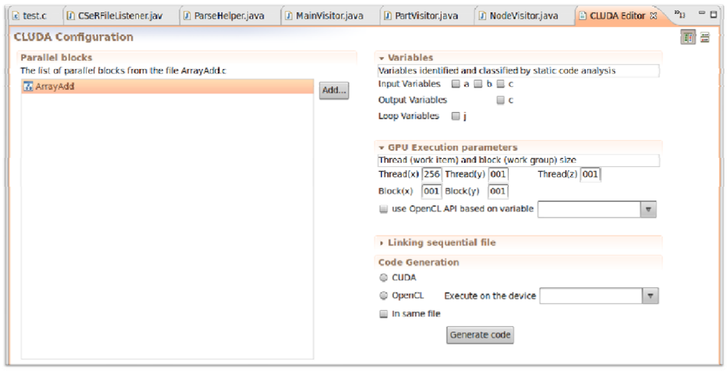
Extending to shared memory: From the GPU examples, any GPU call could be considered as a three-step process: copy or map the variables before the execution, execute on the GPU and copy back or unmap the variables after the execution. An abstract representation of the three steps are: 1) `copyin(vars1)` 2) `callkernel(vars2)` and 3) `copyout(vars3)`. In the code, `vars1,vars2,vars3` refers to the list of variables that have to be copied to the GPU before execution, list of variables required for the call, and list of variables that have to be copied back to the CPU, respectively. In the case of OpenMP programs, the copy operations are not necessary as the execution is on the CPU itself and calling the kernel needs to be aware of the number of threads in which it has to be executed. The abstract API to make a shared memory call is `callParallel(original paramlist, num threads)`.

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



Configuring Execution of `ArrayAdd` using CUDACL

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.

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.

```
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`