Extending Abstract GPU APIs to Shared Memory

SPLASH Student Research Competition
October 19, 2010

Ferosh Jacob
University of Alabama
Department of Computer Science
fjacob@crimson.ua.edu
http://cs.ua.edu/graduate/fjacob
Parallel programming challenges

Duplicated code
“oclMatrVecMul from the OpenCL installation package of NVIDIA, three steps – 1) creating the OpenCL context, 2) creating a command queue and 3) setting up the program – are achieved with 34 lines of code.”

Lack of Abstraction
The programmers should follow a problem-oriented approach rather than the current machine or architecture-oriented approach towards parallel problems.

Performance Evaluation
To make sure the obtained performance cannot be further improved, a program may need to be rewritten to different parallel libraries supporting various approaches (shared memory, GPUs, MPI)
Is it possible to express parallel programs in a platform-independent manner?
Solution approach

1. **AbstractAPIs:** Design a DSL that can express two leading GPU programming languages
   - Support CUDA and OpenCL
   - Automatic data transfer
   - Programmer freed from device variables

2. **CUDACL:** Introduce a configurable mechanism through which programmers fine-tune their parallel programs
   - Eclipse plugin for configuring GPU parameters
   - Supports C (CUDA and OpenCL) and Java (JCUDA, JOCL)
   - Capable of specifying interactions between kernels

3. **CalCon:** Extends our DSL to shared memory; such that programs can be executed on a CPU or GPU
   - Separating problem and configuration
   - Support Fortran and C

4. Extend CalCon to a multi-processor using a Message Passing Library (MPL)
Phase 1: Abstract APIs

Design a DSL that can express two leading GPU programming languages

**API comparison of CUDA and OpenCL**

<table>
<thead>
<tr>
<th>Function</th>
<th>CUDA</th>
<th>OpenCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allocate Memory</td>
<td>cudaMalloc</td>
<td>clCreateBuffer</td>
</tr>
<tr>
<td>Transfer Memory</td>
<td>cudaMemcpy</td>
<td>clReadBuffer</td>
</tr>
<tr>
<td></td>
<td>cudaMemcpy</td>
<td>clWriteBuffer</td>
</tr>
<tr>
<td>Call Kernel</td>
<td>&lt;&lt;&lt; x, y &gt;&gt;&gt;</td>
<td>clEnqueueNDRange</td>
</tr>
<tr>
<td>Block Identifier</td>
<td>blockIdx</td>
<td>get_group_id</td>
</tr>
<tr>
<td>Thread Identifier</td>
<td>threadIdx</td>
<td>get_local_id</td>
</tr>
<tr>
<td>Release Memory</td>
<td>cudaFree</td>
<td>clReleaseMemObject</td>
</tr>
</tbody>
</table>

**LOC comparison of CUDA, CPP and Abstract API**

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>Application</th>
<th>CUDA LOC</th>
<th>CPP LOC</th>
<th>Abstract LOC</th>
<th>#variables reduced</th>
<th>#lines reduced</th>
<th>API usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Vector Addition</td>
<td>29</td>
<td>15</td>
<td>13</td>
<td>3</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Matrix Multiplication</td>
<td>28</td>
<td>14</td>
<td>12</td>
<td>3</td>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Scan Test Cuda</td>
<td>82</td>
<td>NA</td>
<td>72</td>
<td>1</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Transpose</td>
<td>39</td>
<td>17</td>
<td>26</td>
<td>2</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Template</td>
<td>25</td>
<td>13</td>
<td>13</td>
<td>2</td>
<td>12</td>
<td>6</td>
</tr>
</tbody>
</table>
Phase 2: CUDACL

Introduce an easily configurable mechanism through which programmers fine-tune their parallel programs
Phase 3: CalCon

Extend our DSL to shared memory such that programs can be executed on a CPU or GPU

Design details of CalCon

Parallel function calls + configuration

Generating abstract API

CDT parsing and refactoring

Abstract API

pre-defined functions

Generating source code

CDT parsing and refactoring

CUDA code

OpenCL code

OpenMP code
Related works

- **GPU languages**
  - Cg
  - Brook

- **CUDA abstractions**
  - hiCUDA
  - CUDA-lite
  - PGI compiler
  - CuPP framework

- **OpenCL**
  - CalCon

- **Other works**
  - Concurrencer
  - Habenero project
  - Sequoia

**Hardware details or lightweight communication**

**Only tool which supports CUDA, OpenCL, and Shared memory**

**Not portable; Only applicable for GPUs from NVIDIA**
Example: Matrix Transpose

Matrix Transpose (CUDA kernel)

```c
__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];
    }
}
```
Matrix Transpose (OpenMP)

```c
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];
            }
}
```
Matrix Transpose (CalCon)

//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

Data Flow in GPU
42 CUDA kernels were selected from 25 programs.

Program analysis
15 OpenCL programs

Shared memory
10 OpenMP programs from varying domains

http://cs.ua.edu/graduate/fjacob/software/analysis/
Conclusion and Future work

1. Abstract APIs can be used for abstract GPU programming which currently generate CUDA and OpenCL code.
   - 42 CUDA kernels from different problem domains were selected to identify the data flow
   - 15 OpenCL programs were selected to compare with their CUDA counterpart to provide proper abstraction
   - Focus on essence of parallel computing, rather than language-specific accidental complexities of CUDA or OpenCL
   - CUDACL can be used to configure the GPU parameters separate from the program expressing the core computation

2. Extend our DSL to shared memory; such that programs can be executed on a CPU or GPU CalCon
   - Separating problem and configuration
   - Support Fortran and C

3. Extend the DSL to a multi-processor using a Message Passing Library (MPL)
References


Questions?

http://cs.ua.edu/graduate/fjacob/
## OpenMP FORTRAN programs

<table>
<thead>
<tr>
<th>No</th>
<th>Program Name</th>
<th>Total LOC</th>
<th>Parallel LOC</th>
<th>No. of blocks</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2D Integral with Quadrature rule</td>
<td>601</td>
<td>11 (2%)</td>
<td>1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Linear algebra routine</td>
<td>557</td>
<td>28 (5%)</td>
<td>4</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>Random number generator</td>
<td>80</td>
<td>9 (11%)</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Logical circuit satisfiability</td>
<td>157</td>
<td>37 (18%)</td>
<td>1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Dijkstra’s shortest path</td>
<td>201</td>
<td>37 (18%)</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Fast Fourier Transform</td>
<td>278</td>
<td>51 (18%)</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Integral with Quadrature rule</td>
<td>41</td>
<td>8 (19%)</td>
<td>1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Molecular dynamics</td>
<td>215</td>
<td>48 (22%)</td>
<td>4</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>9</td>
<td>Prime numbers</td>
<td>65</td>
<td>17 (26%)</td>
<td>1</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Steady state heat equation</td>
<td>98</td>
<td>56 (57%)</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
Refined FORTRAN code (OpenMP)

! Refined FORTRAN program
call parallel(instance_num,'satisfiability')

ilo2 = ( ( instance_num - id ) * ilo &
   + ( id ) * ihi ) &
   / ( instance_num )

ihi2 = ( ( instance_num - id - 1 ) * ilo &
   + ( id + 1 ) * ihi ) &
   / ( instance_num )

solution_num_local = 0
do i = ilo2, ihi2 - 1
   call i4_to_bvec ( i, n, bvec )
   value = circuit_value ( n, bvec )
   if ( value == 1 ) then
      solution_num_local = solution_num_local + 1
   end if
end do

solution_num = solution_num + solution_num_local
call parallelend('satisfiability')

! Configuration file for FORTRAN program above
block 'satisfiability'
init:
!$omp parallel &
!$omp shared ( lilo, ilo, thread_num ) &
!$omp private ( bvec, id, ilo2, ihi2, j, solution_num_local, value ) &
!$omp reduction ( + : solution_num ).
final:. 
FORTRAN code (MPI)

!Part 1: Master process setting up the data
if ( my_id == 0 ) then    do p = 1, p_num - 1
    my_a = ( real ( p_num - p, kind = 8 ) * a &
      + real ( p - 1, kind = 8 ) * b ) &
      / real ( p_num - 1, kind = 8 )
    target = p
    tag = 1
    call MPI_Send ( my_a, 1, MPI_DOUBLE_PRECISION, &
      target, tag, &MPI_COMM_WORLD, &
      error_flag )
end do
!Part 2: Parallel execution
else
    source = master
    tag = 1
    call MPI_Recv ( my_a, 1, MPI_DOUBLE_PRECISION, source,  tag, &
      MPI_COMM_WORLD, status, error_flag )
    my_total = 0.0D+00
    do i = 1, my_n
      x = ( real ( my_n - i, kind = 8 ) * my_a &
      + real ( i - 1, kind = 8 ) * my_b ) &
      / real ( my_n - 1, kind = 8 )
      my_total = my_total + f ( x )
    end do
    my_total = ( my_b - my_a ) * my_total / real ( my_n, kind = 8 )
end if
!Part 3: Results from different processes are collected to
calculate the final result
    call MPI_Reduce ( my_total, total, 1, &
      MPI_DOUBLE_PRECISION, & MPI_SUM, &
      master, MPI_COMM_WORLD, error_flag)
Refined FORTRAN code (MPI)

! Work share part
  do p = 1, instance_num - 1
    my_a = ( real ( instance_num - p, kind = 8 ) * a &
             + real ( p - 1, kind = 8 ) * b ) &
             + real ( instance_num - 1, kind = 8 )
    call distribute (my_a)
  end do

! Declaring parallel block
  call parallel(num,'quadrature')
  my_total = 0.0D+00
  do i = 1, my_n
    x = ( real ( my_n - i, kind = 8 ) * my_a &
         + real ( i - 1, kind = 8 ) * my_b ) &
         + real ( my_n - 1, kind = 8 )
    my_total = my_total + f ( x )
  end do
  my_total = ( my_b - my_a ) * my_total / real ( my_n, kind = 8 )
  call endparallel('quadrature');

! Configuration file for FORTRAN program above
!
block 'quadrature'
  init:
    source = master
    tag = 1
    call MPI_Recv ( my_a, 1, MPI_DOUBLE_PRECISION, source, &
                   tag, &MPI_COMM_WORLD, status, error_flag ).
  final:
    call MPI_Reduce ( my_total, total, 1, &
                     MPI_DOUBLE_PRECISION, & MPI_SUM, &
                     master, MPI_COMM_WORLD, error_flag ).
  distribute param:
    call MPI_Send ( param, 1, MPI_DOUBLE_PRECISION, &
                   target, tag, &MPI_COMM_WORLD, &
                   error_flag ).
Parallel and OpenMP features

<table>
<thead>
<tr>
<th>Shared memory features</th>
<th>Parallel features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable modifiers, Critical and Singular blocks, Number of threads</td>
<td>Parallel blocks, Reduction and Barrier blocks, Number of instances, Workshare</td>
</tr>
</tbody>
</table>