THE UNIVERSITY OF ALABAMA ENGINEERING

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

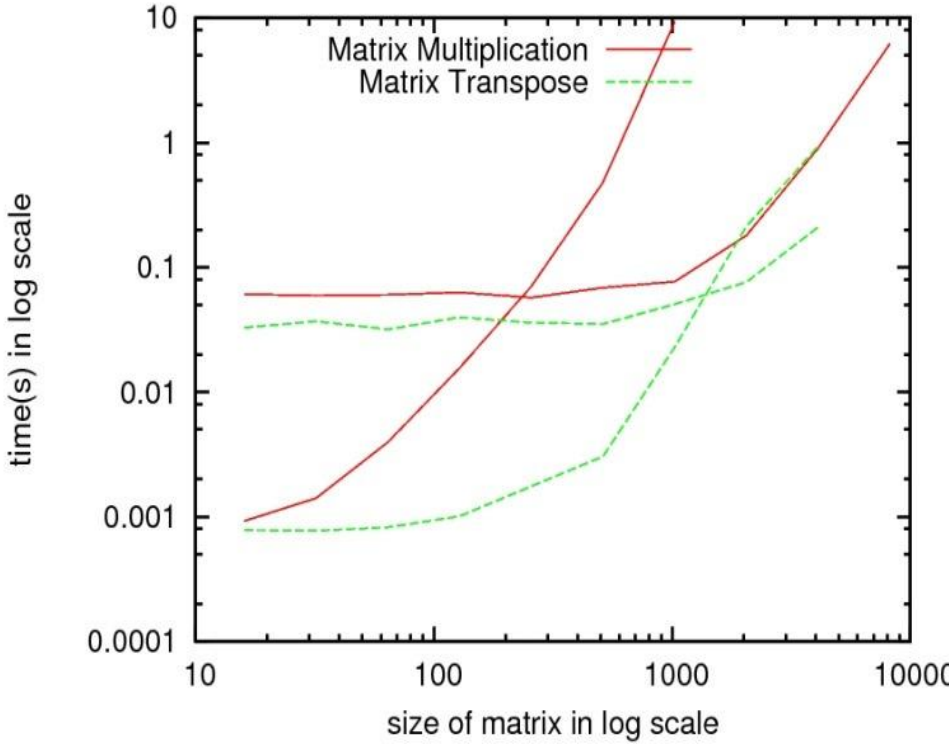
"`oclMatrVecMul` from the OpenCL installation package of NVIDIA, three steps – 1) creating the OpenCL context, 2) creating a command queue and 3) setting up the program – are achieved with 34 lines of code."

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)



Research question

CUDA

p-threads

OpenMPI

OpenCL

OpenMP

Cg

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

Function	CUDA	OpenCL
Allocate Memory	cudaMalloc	clCreateBuffer
Transfer Memory	cudaMemcpy	clReadBuffer clWriteBuffer
Call Kernel	<<< x , y >>>	clEnqueueNDRange clSetKernelArg
Block Identifier	blockIdx	get_group_id
Thread Identifier	threadIdx	get_local_id
Release Memory	cudaFree	clReleaseMemObject

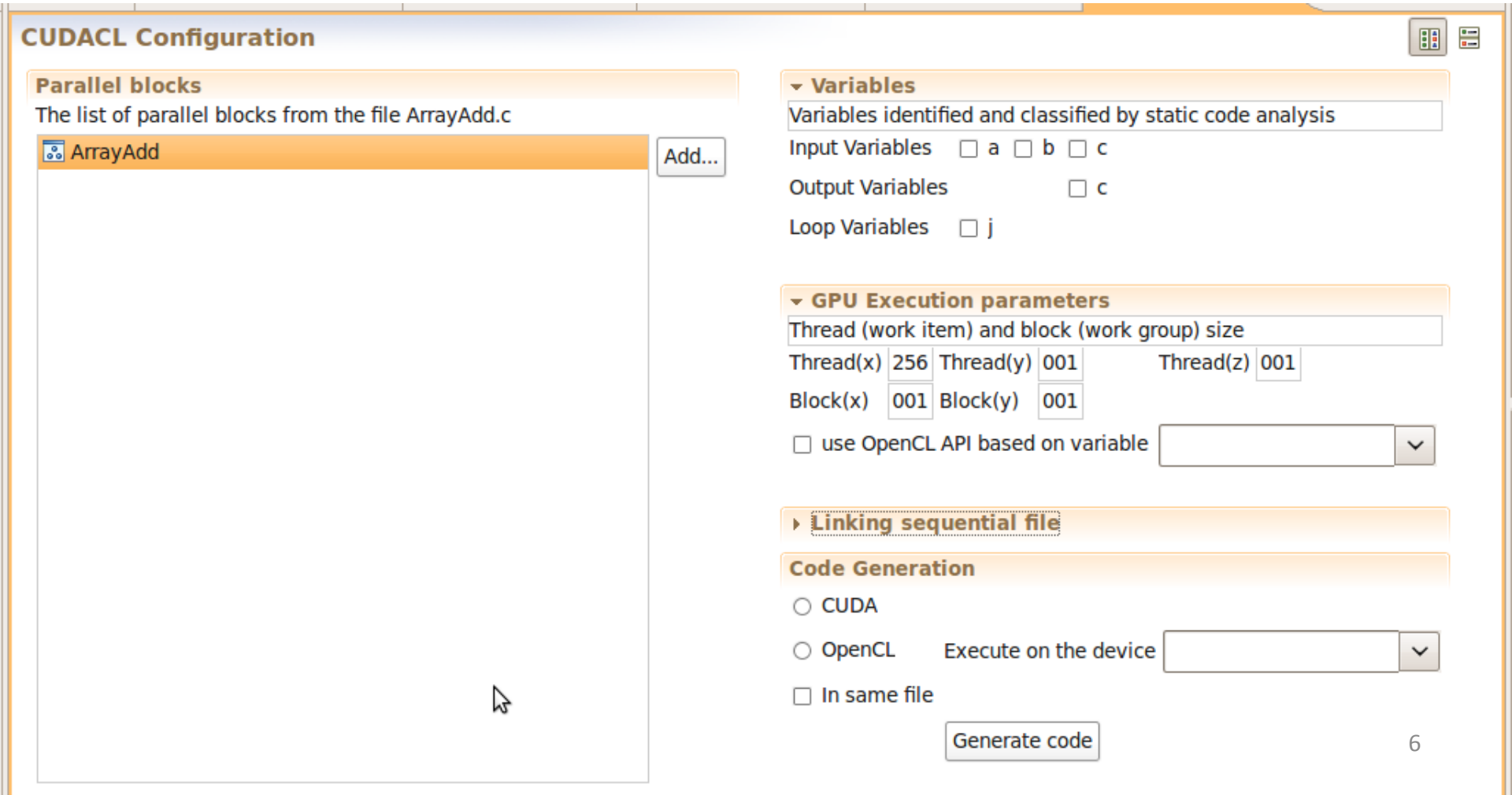
- **XPUmalloc**
- **GPUcall**
- **XPUrelease**
- **GPUinit**

LOC comparison of CUDA, CPP and Abstract API

Sr. No	Application	CUDA LOC	CPP LOC	Abstract LOC	#variables reduced	#lines reduced	API usage
1	Vector Addition	29	15	13	3	16	6
2	Matrix Multiplication	28	14	12	3	14	6
3	Scan Test Cuda	82	NA	72	1	10	12
4	Transpose	39	17	26	2	13	8
5	Template	25	13	13	2	12	6

Phase 2: CUDACL

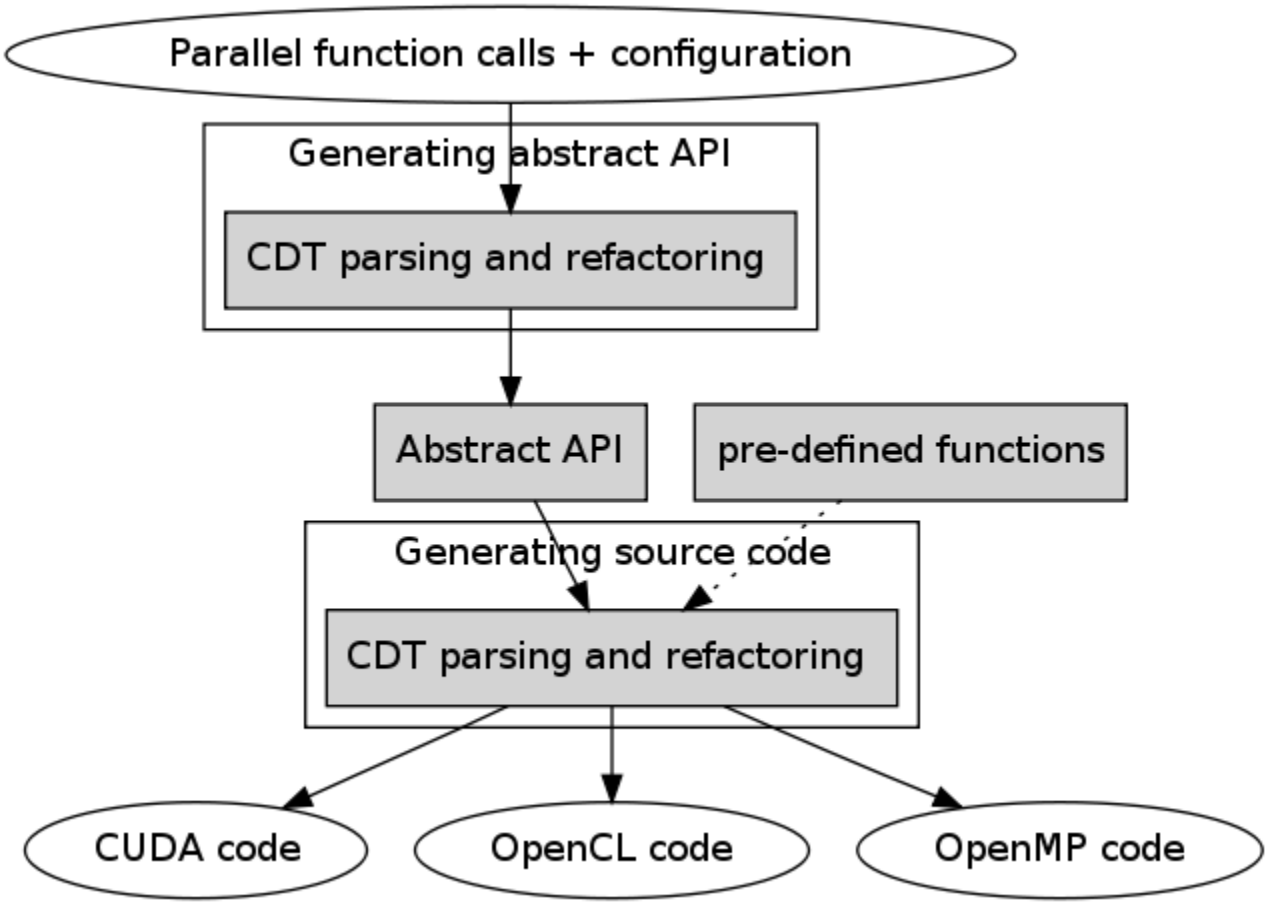
Introduce an easily configurable mechanism through which programmers fine-tune their parallel programs



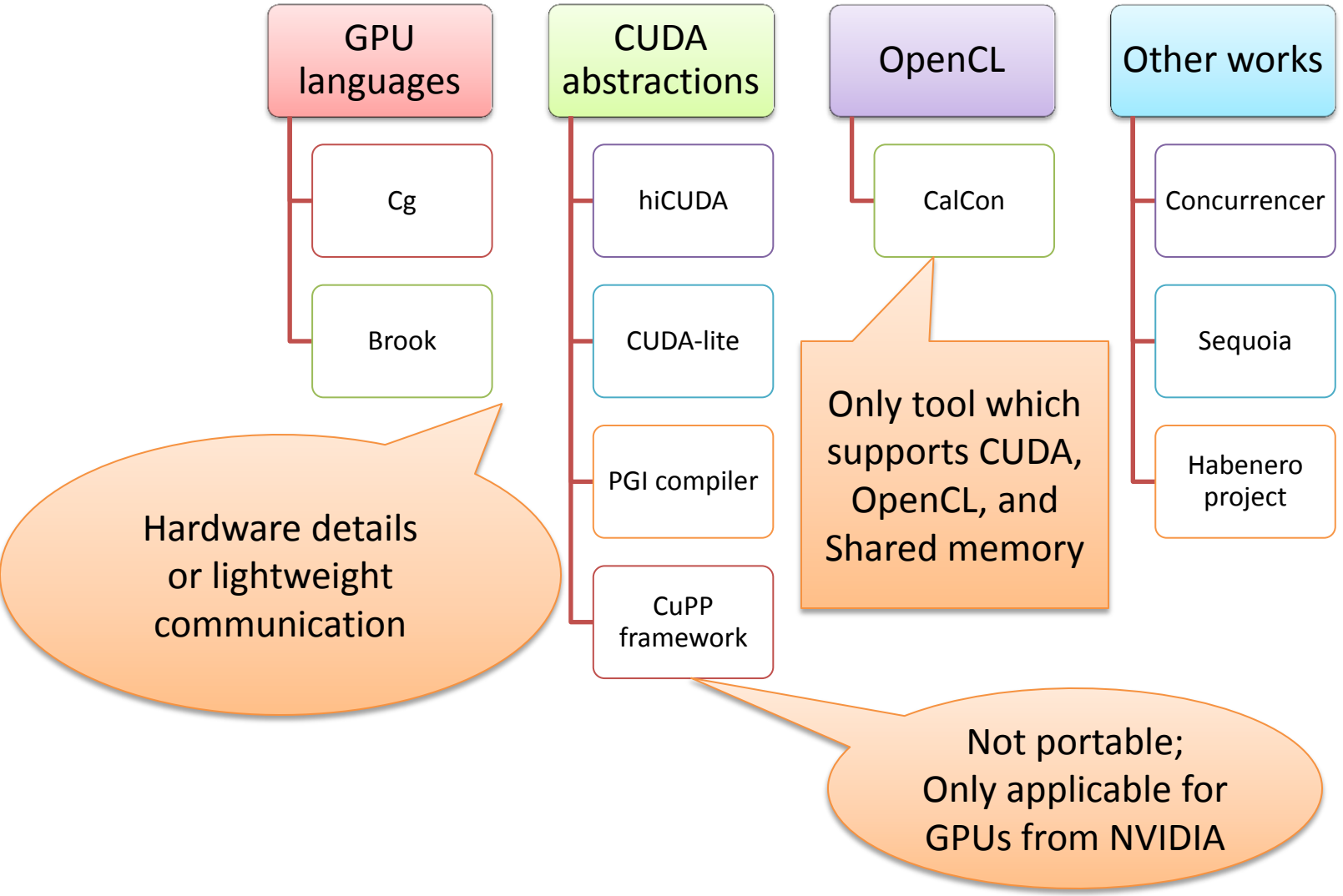
Phase 3: CalCon

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

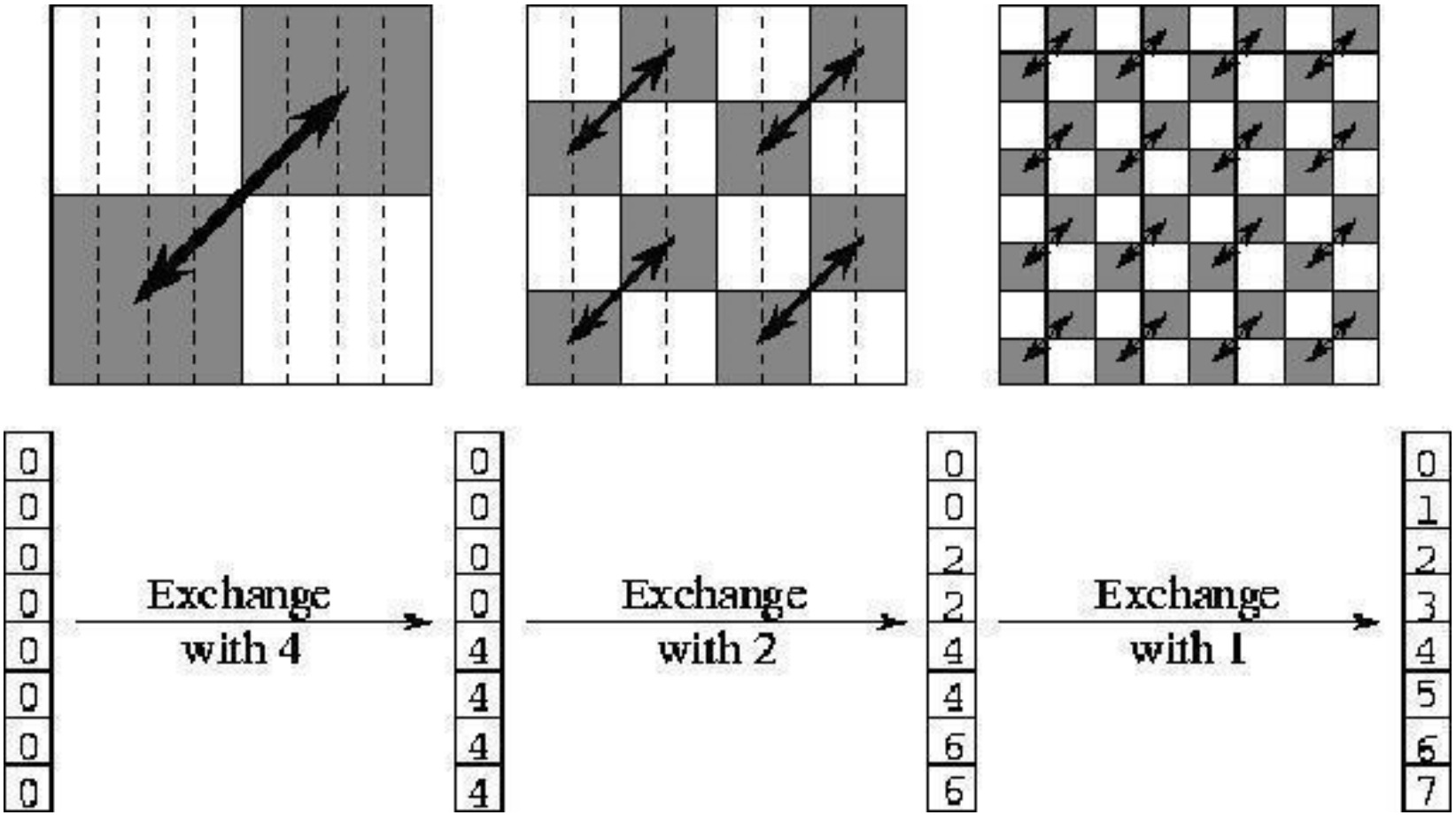
Design details of CalCon



Related works



Example: Matrix Transpose



http://biomatics.org/index.php/Image:Hct.jpg

Matrix Transpose (CUDA kernel)

```
1  __global__ void transpose(float *odata,
2                            float* idata,
3                            int width,
4                            int height){
5    int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
6    int yIndex = blockDim.y * blockIdx.y + threadIdx.y;
7
8     if (xIndex < width && yIndex < height){
9         int index_in  = xIndex + width * yIndex;
10        int index_out = yIndex + height * xIndex;
11        odata[index_out] = idata[index_in];
12    }
13 }
```

Matrix Transpose (OpenMP)

```
1   void transpose(float *odata,
2                  float* idata,
3                  int width,
4                  int height){
5  #pragma omp parallel  private(xIndex,yIndex)
6                        num_threads(N)
7                        default(shared){
8  #pragma omp for
9     for(int xIndex = 0; xIndex < width; xIndex++)
10          for(int yIndex = 0; yIndex < height; yIndex++) {
11             int index_in  = xIndex + width * yIndex;
12             int index_out = yIndex + height * xIndex;
13             odata[index_out] = idata[index_in];
14          }
15   }
16 }
```

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 counter part 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

1. Ferosh Jacob, David Whittaker, Sagar Thapaliya, Purushotham Bangalore, Marjan Mernik, and JeffGray, “CUDACL: A tool for CUDA and OpenCL programmers,” in Proceedings of 17th InternationalConference on High Performance Computing, Goa, India, December 2010, 11 pages.
2. Ferosh Jacob, Ritu Arora, Purushotham Bangalore, Marjan Mernik, and Jeff Gray, “Raising the level of abstraction of GPU-programming,” in Proceedings of the 16th International Conference on Parallel and Distributed Processing, Las Vegas, NV, July 2010, pp. 339-345
3. Ferosh Jacob, Jeff Gray, Purushotham Bangalore, and Marjan Mernik, “Refining High Performance FORTRAN Code from Programming Model Dependencies” HIPC Student Research Symposium, Goa, India, December 2010, 5 pages..

Questions ?

http://cs.ua.edu/graduate/fjacob/

OpenMP FORTRAN programs

No	*Program Name*	*Total LOC*	*Parallel LOC*	*No. of blocks*	*R*	*W*
1	2D Integral with Quadrature rule	601	11 (2%)	1	√	
2	Linear algebra routine	557	28 (5%)	4		√
3	Random number generator	80	9 (11%)	1		
4	Logical circuit satisfiability	157	37 (18%)	1	√	
5	Dijkstra's shortest path	201	37 (18%)	1		
6	Fast Fourier Transform	278	51 (18%)	3		
7	Integral with Quadrature rule	41	8 (19%)	1	√	
8	Molecular dynamics	215	48 (22%)	4	√	√
9	Prime numbers	65	17 (26%)	1	√	
10	Steady state heat equation	98	56 (57%)	3	√ √	

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 ( ihi, ilo, thread_num ) &
!$omp private ( bvec, i, id, ilo2, ihi2,
                 j, solution_num_local, value ) &
!$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

Shared memory features	*Parallel features*
Variable modifiers, Critical and Singular blocks, Number of threads	Parallel blocks, Reduction and Barrier blocks, Number of instances, Workshare