GPU programming made easier

Jacob Jepsen

6. June 2014

University of Copenhagen

Department of Computer Science

Introduction

- We created a tool that reduces the development time of GPU code.
- The input is a loop nest which is parsed into an internal representation.
- We generate code which makes the loop executable on a GPU.
- We apply optimizations to the code and perform benchmarks on CPU and GPU architectures.
- Our code is 2-258X faster than code generated by an OpenACC compiler, 1-37X faster than optimized CPU code, and attains up to 25% of peak performance.

Ideas

- We want to reduce errors and the development time, while ensuring high performance.
- Optimizations on OpenCL code are regular and the same optimization can be applied to many different pieces of code.
- A tool with a catalogue of optimizations which can be performed semi-automatically by the programmer.
- Move toward fully-automatic optimizations to make the tool usable for novices in GPU programming.

Overview

- Front end and code generation
- OpenCL background
- GPU background
- Transformations
- Pattern-matching rules
- Performance experiments
- Conclusion

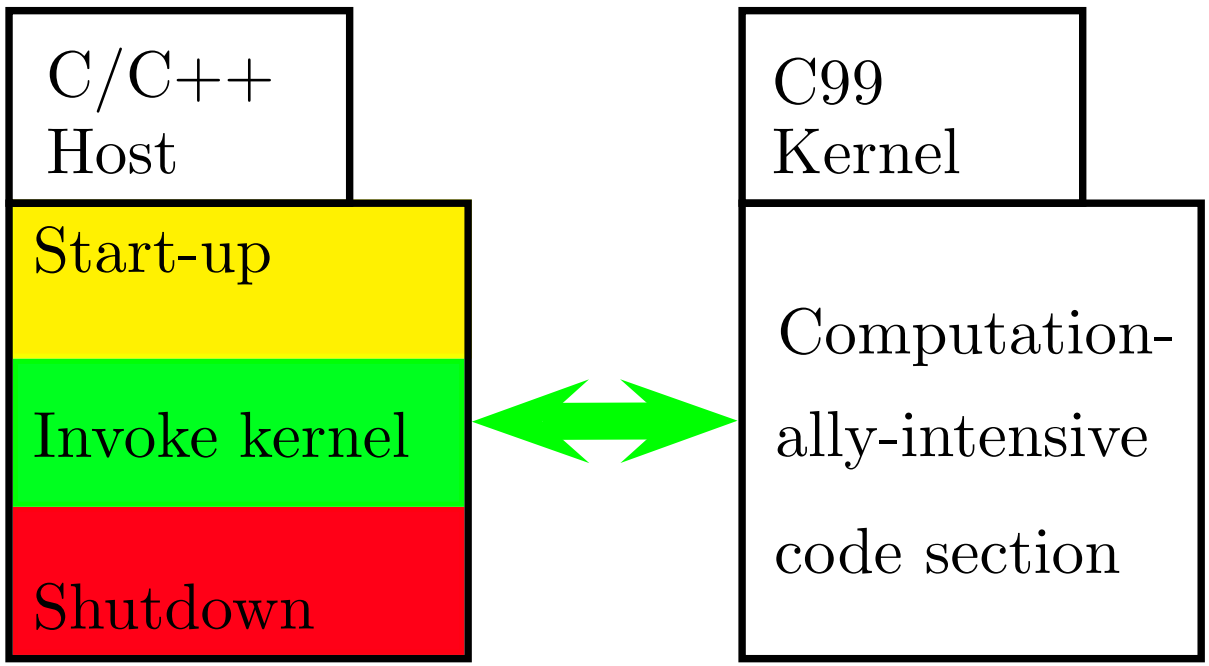
Front end and code generation


```
t_PLUS = r'\+'

def p_for_loop(p):
    """ for_loop : FOR LPAREN assignment_expression SEMI binop SEMI
                                 increment RPAREN compound
    """
    p[0] = ForLoop(p[3], p[5], p[7], p[9])

class ForLoop(Node):
    def __init__(self, init, cond, inc, compound):
        self.init = init
        self.cond = cond
        self.inc = inc
        self.compound = compound
```

OpenCL background



- The host code sets up data structures and manages the GPU execution.

GPU (GK110) background

- Warps: 32 threads which execute the same instructions in a Single Instruction Multiple Threads (SIMT) fashion.
- Registers: 255 private to each thread. Use as cache for data with time locality.
- Local memory: scratchpad shared between local work group. Effective when data is shared/broadcasted.
- Memory coalescing: data accessed by threads with consecutive thread IDs should be located consecutively in memory.

Transformations

- DefineArg: Similar to constant propagation, we can compile the values of variables into the kernel code, in order to allow the compiler to do more optimizations.
- Transposition: We transpose the data in an array in order to create coalesced memory access.

HoistToReg and HoistToRegLoop

- Read data once, save it in registers, and reread the data from there, similar to loop-invariant code motion.

Original code

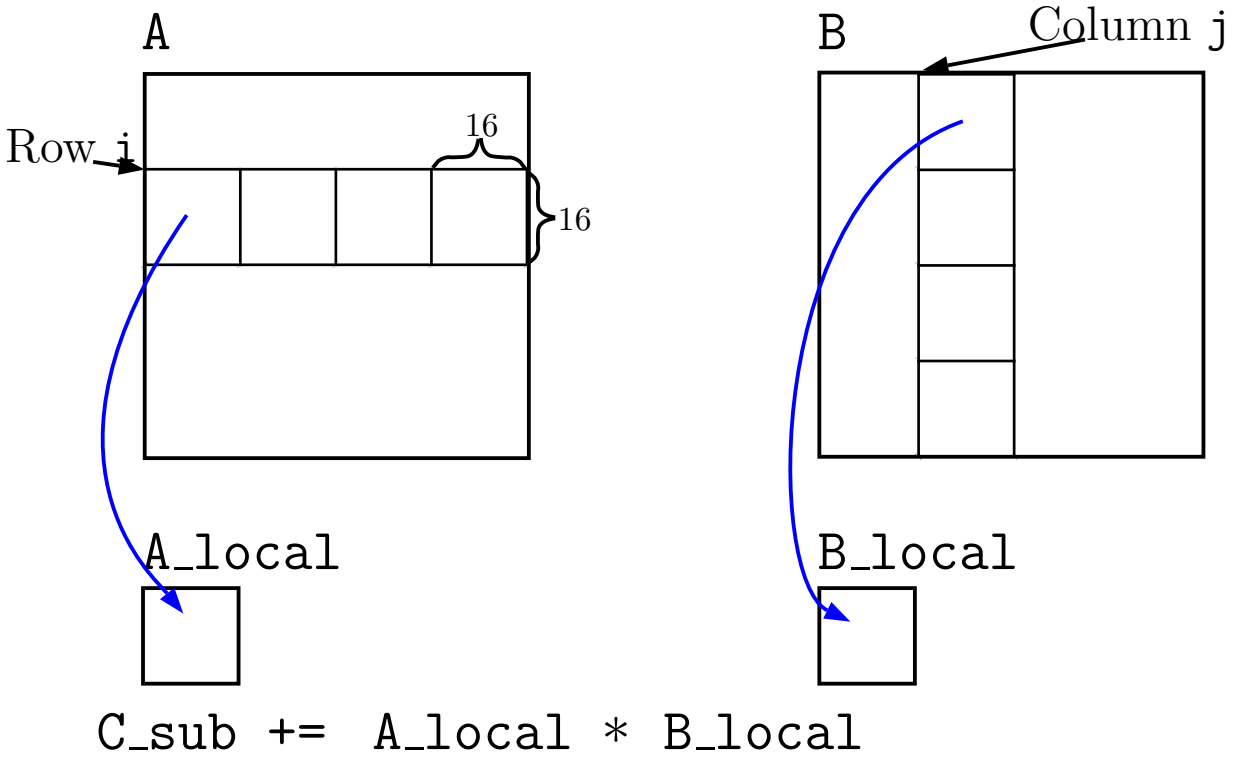
```
for (unsigned j = 0; j < N; j++) {
    float a_x = Pos[0][get_global_id(0)];
    float a_y = Pos[1][get_global_id(0)];
    float a_m = Mas[get_global_id(0)];
    ...
}
```

Transformed code

```
float Mas0_reg = Mas[get_global_id(0)];
float Pos0_reg = Pos[0][get_global_id(0)];
float Pos1_reg = Pos[1][get_global_id(0)];
for (unsigned j = 0; j < N; j++) {
    float a_x = Pos0_reg;
    float a_y = Pos1_reg;
    float a_m = Mas0_reg;
    ...
}
```

TileInLocal

- Load shared data into local memory once and let each thread read the data they need from local memory.



TileInLocal

Original code

```
float C_sub = 0;
for (unsigned k = 0; k < wA; k++) {
    C_sub += A[get_global_id(1)][k] * B[k][get_global_id(1)];
}
C[get_global_id(1)][get_global_id(0)] = C_sub;
```

Transformed code

```
1  float C_sub = 0;
2  for (unsigned k = 0; k < wA; k+=16) {
3      A_local[get_local_id(1)][get_local_id(0)] =
4                                  A[get_global_id(1)][k + get_local_id(0)];
5      B_local[get_local_id(1)][get_local_id(0)] =
6                                  B[k + get_local_id(1)][get_global_id(0)];
7      barrier(CLK_LOCAL_MEM_FENCE);
8      for (unsigned kk = 0; kk < 16; kk++) {
9          C_sub += A_local[get_local_id(1)][kk] *
10                                 B_local[kk][get_local_id(0)];
11     }
12     barrier(CLK_LOCAL_MEM_FENCE);
13 }
14 C[get_global_id(1)][get_global_id(0)] = C_sub;
```

Pattern matching

- We link each transformations to a pattern. The presence of the pattern in the code, means that the linked transformation is applicable.
- We iterate over the array references and search for patterns. For each found pattern we check a set of conditions, and if met, we perform the linked transformation.
- The conditions are not exhaustive, but sufficiently thorough to make them usable in practice.
- The running time is linear in the number of array references.

DefineArg and Transposition

- For DefineArg we do no pattern matching, and perform the transformation always.
- For Transposition we divide the pattern-matching rule into two cases: 1D- and 2D-parallelization.

For 1D:

```
A[get_global_id(0)][d]
```

For 2D:

```
A[get_global_id(0)][get_global_id(1)]
```

HoistToReg and HoistToRegLoop

- For HoistToReg: an array reference that is inside one or more loops, but contains no loop index.
- For HoistToRegLoop: an array reference that is inside two loops, and the loop index of the outermost loop is not in the subscript of the reference, but the loop index of the innermost loop is.
- We use at most 20 registers.
- We decide at run-time whether to include the transformation.

For HoistToReg

```
for (unsigned k = 0; k < N; k++) {
     ... = A[10];
     ... = B[get_global_id(0)][l];
     for (unsigned g = 0; g < dim; g++) {
          ... = C[get_global_id(1)];
          ... = D[l][10];
     }
}
```

For HoistToRegLoop

```
for (unsigned k = 0; k < N; k++) {
     for (unsigned g = 0; g < dim; g++) {
          ... = A[10][g];
          ... = B[g][get_global_id(0)];
          ... = C[get_global_id(1)][g];
          ... = D[g][l];
     }
}
```

TileInLocal

- An array with two subscripts where one contains a loop index and the other a global thread identifier.
- Additional conditions:
  - The loop index must have a stride of one.
  - The number of loop iterations must be divisable by a tiling factor.
- Check last condition at run-time.

```
for (unsigned k = 0; k < N; k++) {
      ... = A[get_global_id(1)][k];
      ... = B[k][get_global_id(0)];
}
```

Performance experiments

We compare the performance against:

1. Frameworks with comparative capabilities
2. The theoretical peak performance of the test hardware
3. The performance of CPUs

- We found one framework, the OpenACC API, which has similar capabilities as our tool.
- We extended our tool to generate optimized code for CPUs.
- The benchmarks were run on an NVIDIA K20 GPU, and a machine with two Intel Xeon E5-2670 clocked at 2.6 GHz.

Performance experiments (2)

We have a mix of programs: compute/memory bound, small/high N.

	MatMul	Squared Euclid	NBody	Laplace	Gaussian kernels	Jacobi
DefineArg	×	×	×	×	×	×
Transposition		×		×		
HoistToReg			×			
HoistToRegLoop		×		×		
TileInLocal	×				×	
TileInLocalStencil						×

Table: Applicability of the transformations.

Performance experiments (3)

	MatMul	Jacobi	Squared Euclid	NBody	Laplace	Gaussian kernels
GPU Optimized to GPU Basic	3.1	1	55.7	3.4	3.6	1.7
GPU Basic to PGI	0.9	1.9	4.6	2.2	–	–
GPU Optimized to PGI	2.8	1.9	257.4	7.5	–	–

Table: Speedup in the execution time of the code generated by the different frameworks.

Performance experiments (4)

	MatMul	Jacobi	Squared Euclid	NBody	Laplace	Gaussian kernels
Performance [GFlop/s]	205	4	611	872	245	104
% of peak performance	6	1	18	25	21	3

	MatMul	Jacobi	Squared Euclid	NBody	Laplace	Gaussian kernels
CPU Optimized to CPU Basic	6.8	0.7	1.1	1.1	1.1	15.6
GPU Optimized to CPU Optimized	3.3	0.6	36.1	10.9	6.5	1.8

Conclusion

- Design of a model of how data can be reused.
- We found pattern-matching rules which allow the transformations to be performed automatically.
- Conditions pertaining to the applicability of a transformations needs to be checked at compile time and at run-time.
- Benchmarks show significant improvements, up to one order of magnitude, in time-to-solution when comparing to OpenACC and optimized CPU code.
- For three programs, the generated code attained close to 25% of peak performance of the GPU. For the others, further transformations would be needed to obtain higher performance.
- My paper has been accepted for PSTI 2014.