A Case for Better Integration of Host and Target Compilation When Using OpenCL for FPGAs

Taylor Lloyd, Artem Chikin, Erick Ochoa, Karim Ali, José Nelson Amaral

University of Alberta

University of Alberta Systems Group

- Focused on compiler optimizations, heterogeneous systems
- Recently working primarily on GPU computing

So can traditional compiler techniques help with OpenCL for FPGAs?

Intel® FPGA SDK for OpenCL

Best Practices Guide

UG-OCL003
2017.05.08

Last updated for Intel® Quartus® Prime Design Suite: 17.0

Background: OpenCL Execution Models

Data Parallelism (NDRange)

- kernel defined per-thread
- Kernel execution defines number and grouping of threads
- Behaviour varies by querying thread ID

Task Parallelism (Single Work-Item)

- Kernel defines complete unit of work
- Kernel execution starts single thread

Background: OpenCL Execution Model

NDRange Example

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length) {
  int index = get_global_id(0);
  while (index<length) {
    tgt[index] = src[index];
    index += get_global_size(0);
  }
}
```

Single Work-Item Example

Background: OpenCL Execution Model

NDRange Example

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length) {
  int index = get_global_id(0);
  while (index<length) {
    tgt[index] = src[index];
    index += get_global_size(0);
  }
}
```

```
int offset = 0, threads = 2048, groupsize = 128;
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

Single Work-Item Example

Background: OpenCL Execution Model

NDRange Example

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length) {
  int index = get_global_id(0);
  while (index<length) {
    tgt[index] = src[index];
    index += get_global_size(0);
  }
}
```

```
int offset = 0, threads = 2048, groupsize = 128;
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

Single Work-Item Example

```
__kernel void memcpy(char* tgt,
                    (char* src,
                     int length) {
  for(int i=0; i<length; i++) {
    tgt[i] = src[i];
  }
}
```

Background: OpenCL Execution Model

NDRange Example

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length) {
  int index = get_global_id(0);
  while (index<length) {
    tgt[index] = src[index];
    index += get_global_size(0);
  }
}
```

```
int offset = 0, threads = 2048, groupsize = 128;
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

Single Work-Item Example

```
__kernel void memcpy(char* tgt,
                    (char* src,
                     int length) {
  for(int i=0; i<length; i++) {
    tgt[i] = src[i];
  }
}
```

```
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueTask(
  queue, kernel,
  0, NULL, NULL);
```

Single Work-Item Kernel versus NDRange Kernel

“ Intel recommends that you structure your OpenCL kernel as a single work-item, if possible”[1]

[1]https://www.altera.com/en_US/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf Pg. 7

NDRange Kernel ➡ Single Work Item

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length



                             ) {

    int index = get_global_id(0);
    while (index<length) {
      tgt[index] = src[index];
      index += get_global_size(0);
    }

}
```

NDRange Kernel ➡ Single Work Item

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length,
                     int offset,
                     int threads,
                     int group ) {

    int index = get_global_id(0);
    while (index<length) {
      tgt[index] = src[index];
      index += get_global_size(0);
    }

}
```

NDRange Kernel ➡ Single Work Item

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length,
                     int offset,
                     int threads,
                     int groups) {
  for(int tid=offset; tid<offset+threads; tid++) {
    int index = tid;
    while (index<length) {
      tgt[index] = src[index];
      index += threads;
    }
  }
}
```

Is that really better?

Loop Canonicalization

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length,
                     int offset,
                     int threads,
                     int groups) {
  for(int tid=offset; tid<offset+threads; tid++) {
    int index = tid;
    for (int i=0; i<length/threads; i++) {
      if(index+i*threads < length)
        tgt[index+i*threads] = src[index+i*threads];
    }
  }
}
```

Loop Canonicalization

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length,
                     int offset,
                     int threads,
                     int groups) {
  for(int j=0; j<threads; j++) {
    int tid = j+offset;
    int index = tid;
    for (int i=0; i<length/threads; i++) {
      if(index+i*threads < length)
        tgt[index+i*threads] = src[index+i*threads];
    }
  }
}
```

Loop Collapsing

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length,
                     int offset,
                     int threads,
                     int groups) {
  for(int x=0; x<threads*length/threads; x++) {
    int j = x/(length/threads);
    int i = x%(length/threads);
    int tid = j+offset;
    int index = tid;
    if(index+i*threads < length)
      tgt[index+i*threads] = src[index+i*threads];
    }
  }
}
```

Copy Propagation

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length,
                     int offset,
                     int threads,
                     int groups) {
  for(int x=0; x<length; x++) {
    int j = x/(length/threads);
    int i = x%(length/threads);
    if(j+offset+i*threads < length)
      tgt[j+offset+i*threads] =
          src[j+offset+i*threads];
    }
  }
}
```

Why isn’t this done today?

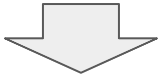
Recall: Host OpenCL API

- Host code must be rewritten to pass new arguments, call different API

Recall: Host OpenCL API

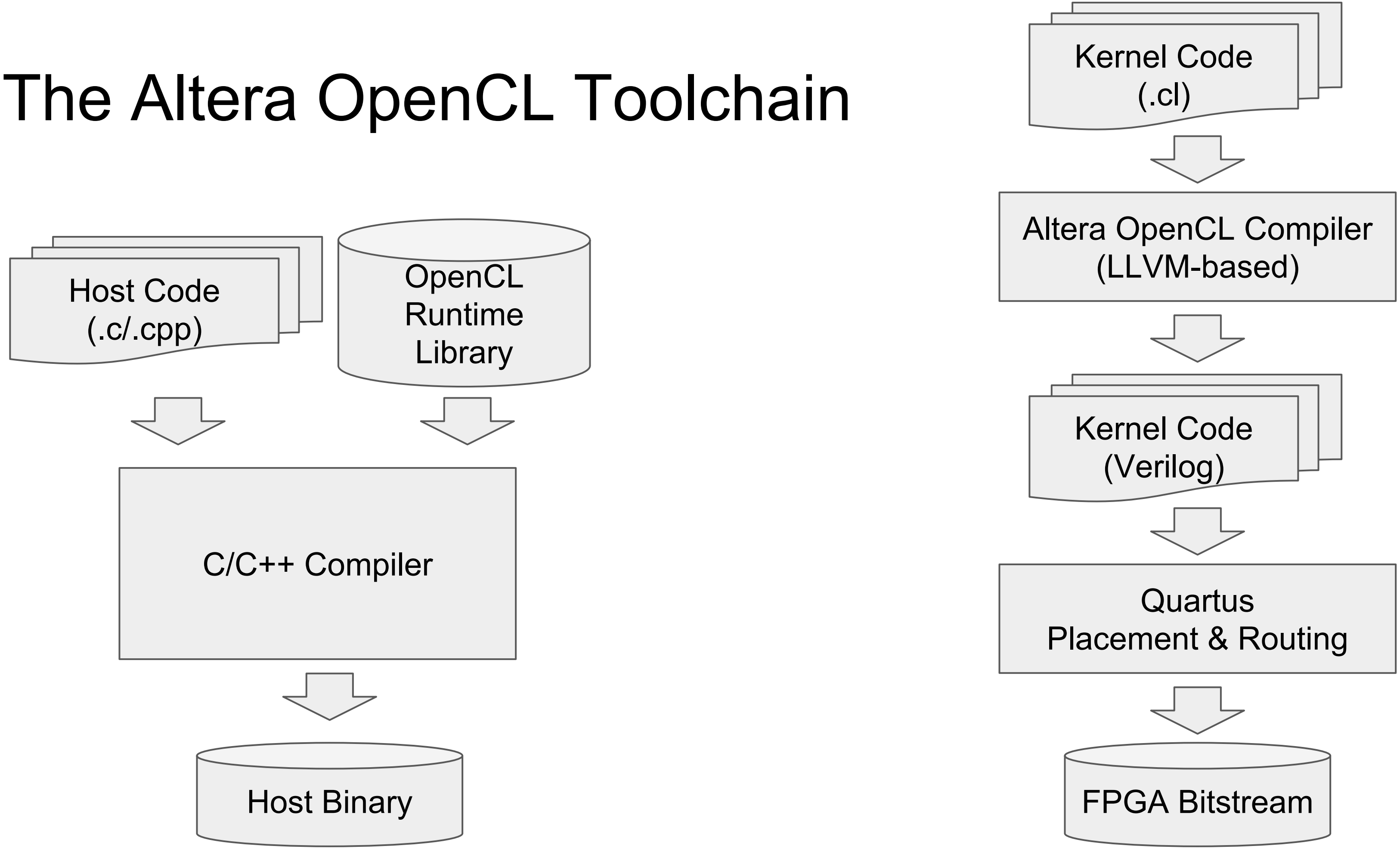
- Host code must be rewritten to pass new arguments, call different API

```
int offset = 0, threads = 2048, groupsize = 128;
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```



```
int offset = 0, threads = 2048, groupsize = 128;
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clSetKernelArg(kernel, 3, sizeof(int), offset);
clSetKernelArg(kernel, 4, sizeof(int), threads);
clSetKernelArg(kernel, 5, sizeof(int), groups);
clEnqueueTask(
  queue, kernel,
  0, NULL, NULL);
```

The Altera OpenCL Toolchain



The Argument for Separation

- Device-side code can be Just-In-Time (JIT) compiled for each device

The Argument for Separation

- Device-side code can be Just-In-Time (JIT) compiled for each device

- Host compilers can be separately maintained by experts (icc, xlc, gcc, clang)

The Argument for Separation

- Device-side code can be Just-In-Time (JIT) compiled for each device
- Host compilers can be separately maintained by experts (icc, xlc, gcc, clang)
- Host code can be recompiled without needing to recompile device code

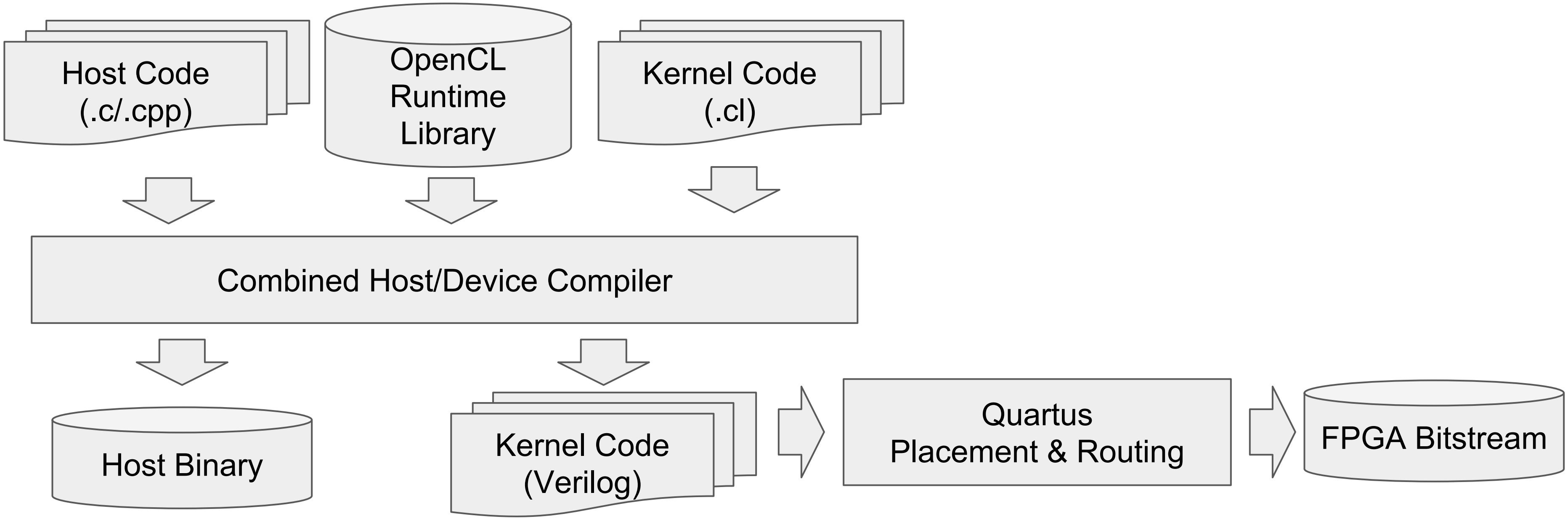
The Argument for Combined Compilation

- Execution context information (constants, pointer aliases) can be passed from host to device

- Context information allows for better compiler transformations (Strength Reduction, Pipelining)

- Better transformations improve final executables

Our Proposed OpenCL Toolchain



Research Question:

Can OpenCL be better targeted to FPGAs given communication between host and device compilers?

Inspiration

Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs

Hamid Reza Zohouri*, Naoya Maruyama†*, Aaron Smith‡,
Motohiko Matsuda† and Satoshi Matsuoka*
*Tokyo Institute of Technology, †RIKEN Advanced Institute for Computational Science, ‡Microsoft Research
Email: *{zohouri.h.aa@m, matsu@is}.titech.ac.jp, †{nmaruyama, m-matsuda}@riken.jp, ‡aaron.smith@microsoft.com

[SC 16]

Inspiration

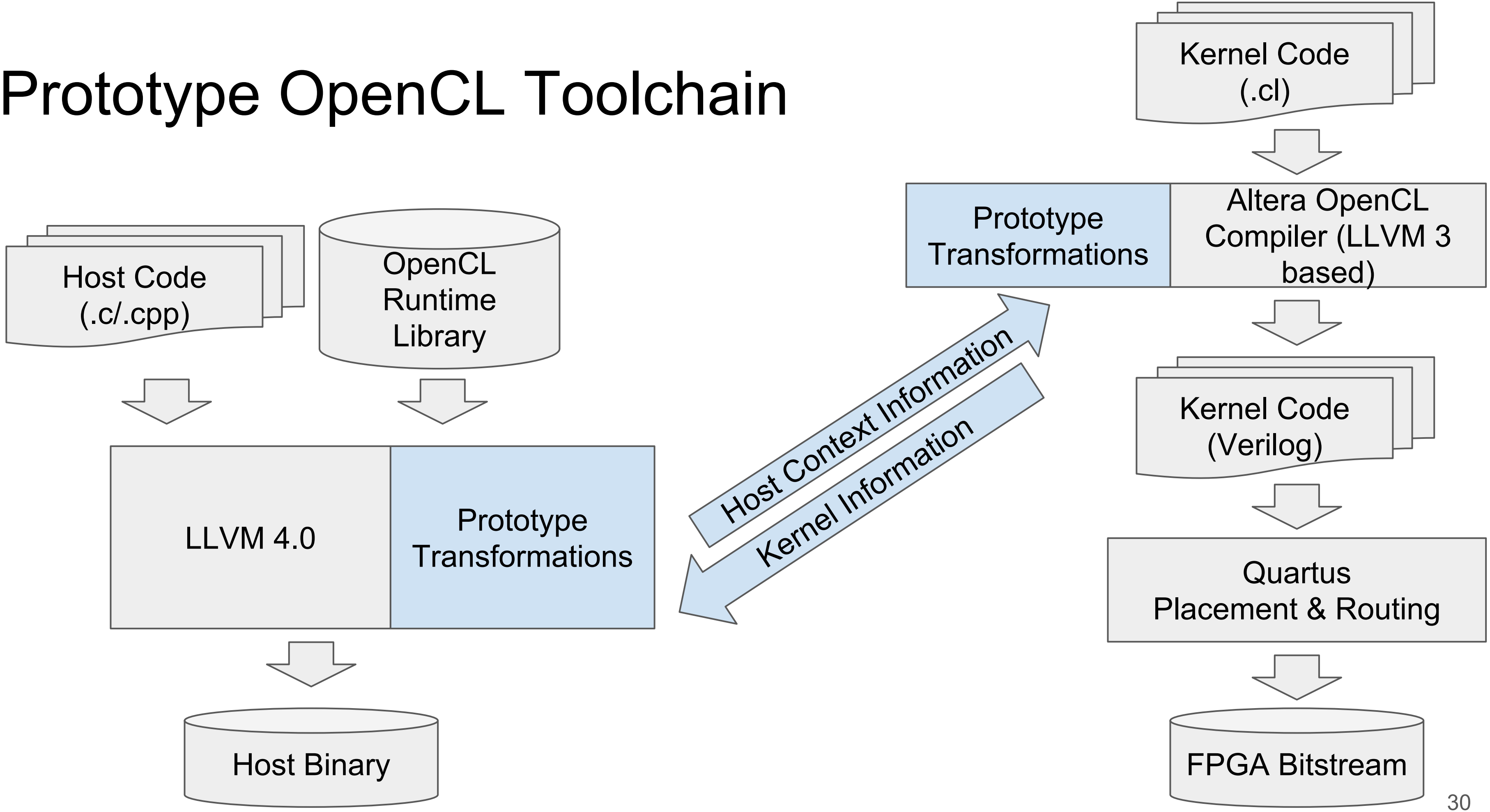
Evaluating and Optimizing OpenCL Kernels for High Performance Computing with FPGAs

Hamid Reza Zohouri[*], Naoya Maruyama[†*], Aaron Smith[‡],
Motohiko Matsuda[†] and Satoshi Matsuoka[*]
[*]Tokyo Institute of Technology, [†]RIKEN Advanced Institute for Computational Science, [‡]Microsoft Research
Email: [*]{zohouri.h.aa@m, matsu@is}.titech.ac.jp, [†]{nmaruyama, m-matsuda}@riken.jp, [‡]aaron.smith@microsoft.com

- Zohouri et al. hand-tuned OpenCL benchmarks for FPGA execution
- Achieved speedups of **30%** to **100x**
- Can we match their performance through compiler transformations?

[SC 16]

Prototype OpenCL Toolchain



Prototype Transformations

1. **Geometry Propagation**
2. NDRange To Loop
3. Restricted Pointer Analysis
4. Reduction Dependence Elimination

1. Geometry Propagation - Motivation

- Operations on constants in kernel can undergo strength reduction

1. Geometry Propagation - Motivation

- Operations on constants in kernel can undergo strength reduction

- Loops of known size are easier to manipulate by the compiler

1. Geometry Propagation

1. Collect Host-Side kernel invocations

```
int offset = 0, threads = 2048, groupsize = 128;
cl_kernel kernel = clCreateKernel(program, “memcpy”, &err);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

1. Geometry Propagation

1. Collect Host-Side kernel invocations

```
int offset = 0, threads = 2048, groupsize = 128;
cl_kernel kernel = clCreateKernel(program, "memcpy", &err);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

1. Geometry Propagation

1. Collect Host-Side kernel invocations
2. Identify associated kernels

```
int offset = 0, threads = 2048, groupsize = 128;
cl_kernel kernel = clCreateKernel(program, “memcpy”, &err);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

1. Geometry Propagation

1. Collect Host-Side kernel invocations
2. Identify associated kernels

```
int offset = 0, threads = 2048, groupsize = 128;
cl_kernel kernel = clCreateKernel(program, “memcpy”, &err);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

1. Geometry Propagation

1. Collect Host-Side kernel invocations
2. Identify associated kernels

```
int offset = 0, threads = 2048, groupsize = 128;
cl_kernel kernel = clCreateKernel(program, “memcpy”, &err);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

1. Geometry Propagation

1. Collect Host-Side kernel invocations
2. Identify associated kernels
3. Identify call geometry

```
int offset = 0, threads = 2048, groupsize = 128;
cl_kernel kernel = clCreateKernel(program, “memcpy”, &err);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

1. Geometry Propagation

1. Collect Host-Side kernel invocations
2. Identify associated kernels
3. Identify call geometry

```
int offset = 0, threads = 2048, groupsize = 128;
cl_kernel kernel = clCreateKernel(program, “memcpy”, &err);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

1. Geometry Propagation

1. Collect Host-Side kernel invocations
2. Identify associated kernels
3. Identify call geometry
4. Discovered constants are passed to the device compiler

```
int offset = 0, threads = 2048, groupsize = 128;
cl_kernel kernel = clCreateKernel(program, “memcpy”, &err);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueNDRangeKernel(
  queue, kernel,
  1, &offset, &threads, &groupsize,
  0, NULL, NULL);
```

Prototype Transformations

1. Geometry Propagation
2. **NDRange To Loop**
3. Restricted Pointer Analysis
4. Reduction Dependence Elimination

2. NDRange To Loop - Motivation

1. Allow threads to be pipelined together, and share intermediate products

2. NDRange To Loop - Motivation

1. Allow threads to be pipelined together, and share intermediate products

2. Enable further optimization: e.g. *Reduction Dependence Elimination*

2. NDRange To Loop - Motivation

1. Allow threads to be pipelined together, and share intermediate products

2. Enable further optimization: e.g. *Reduction Dependence Elimination*

3. Allow inner loops in kernels to be pipelined

2. NDRange To Loop

1. Inject kernel parameters for non-constant geometry

```
__kernel void kernel(...) {
  int index = get_global_id(0);
  f(index);
  barrier(CLK_GLOBAL_MEM_FENCE);
  g(index);
}
```

2. NDRange To Loop

1. Inject kernel parameters for non-constant geometry

```
__kernel void kernel(...,
int dims, int gbl_offset_x,
int gbl_size_x, int lcl_size_x, ...) {
  int index = get_global_id(0);
  f(index);
  barrier(CLK_GLOBAL_MEM_FENCE);
  g(index);
}
```

2. NDRange To Loop

1. Inject kernel parameters for non-constant geometry
2. Detect number of dimensions

```
__kernel void kernel(...,
int dims, int gbl_offset_x,
int gbl_size_x, int lcl_size_x, ...) {
  int index = get_global_id(0);
  f(index);
  barrier(CLK_GLOBAL_MEM_FENCE);
  g(index);
}
```

2. NDRange To Loop

1. Inject kernel parameters for non-constant geometry
2. Detect number of dimensions

```
__kernel void kernel(...,
int dims, int gbl_offset_x,
int gbl_size_x, int lcl_size_x, ...) {
  int index = get_global_id(0);
  f(index);
  barrier(CLK_GLOBAL_MEM_FENCE);
  g(index);
}
```

2. NDRange To Loop

1. Inject kernel parameters for non-constant geometry
2. Detect number of dimensions
3. Identify synchronization points

```
__kernel void kernel(...,
int dims, int gbl_offset_x,
int gbl_size_x, int lcl_size_x, ...) {
  int index = get_global_id(0);
  f(index);
  barrier(CLK_GLOBAL_MEM_FENCE);
  g(index);
}
```

2. NDRange To Loop

1. Inject kernel parameters for non-constant geometry
2. Detect number of dimensions
3. Identify synchronization points

```
__kernel void kernel(...,
int dims, int gbl_offset_x,
int gbl_size_x, int lcl_size_x, ...) {
  int index = get_global_id(0);
  f(index);
  barrier(CLK_GLOBAL_MEM_FENCE);
  g(index);
}
```

2. NDRange To Loop

1. Inject kernel parameters for non-constant geometry
2. Detect number of dimensions
3. Identify synchronization points
4. Wrap unsynchronized portions In loops

```
__kernel void kernel(...,
int dims, int gbl_offset_x,
int gbl_size_x, int lcl_size_x, ...) {
  int index = get_global_id(0);
  f(index);
  barrier(CLK_GLOBAL_MEM_FENCE);
  g(index);
}
```

2. NDRange To Loop

1. Inject kernel parameters for non-constant geometry
2. Detect number of dimensions
3. Identify synchronization points
4. Wrap unsynchronized portions In loops

```
__kernel void kernel(...,
int dims, int gbl_offset_x,
int gbl_size_x, int lcl_size_x, ...) {
  for(int i=0;i<gbl_size_x;i+=lcl_size_x)
    for(int j=0; j<lcl_size_x; j++) {
      int index = i+j;
      f(index);
    }
  }
  for(int i=0;i<gbl_size_x;i+=lcl_size_x)
    for(int j=0; j<lcl_size_x; j++) {
      int index = i+j;
      g(index);
    }
  }
}
```

Prototype Transformations

1. Geometry Propagation
2. NDRange To Loop
3. **Restricted Pointer Analysis**
4. Reduction Dependence Elimination

3. Restricted Pointer Analysis - Motivation

- Pipelining of FPGA loops often fails due to aliased memory operations

3. Restricted Pointer Analysis - Motivation

- Pipelining of FPGA loops often fails due to aliased memory operations

- Marking parameters *restrict* dramatically reduces false aliasing

3. Restricted Pointer Analysis - Motivation

- Pipelining of FPGA loops often fails due to aliased memory operations
- Marking parameters *restrict* dramatically reduces false aliasing
- Detecting non-aliasing parameters must be done through host analysis

3. Restricted Pointer Analysis

1. (Host) Identify pointed-to host buffers

```
__kernel void memcpy(char* tgt,
                     char* src,
                     int length) {
  for(int i=0; i<length; i++) {
    tgt[i] = src[i];
  }
}
```

```
cl_mem srcbuf = clCreateBuffer(...);
cl_mem tgtbuf = clCreateBuffer(...);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueTask(
  queue, kernel,
  0, NULL, NULL);
```

3. Restricted Pointer Analysis

1. (Host) Identify pointed-to host buffers

2. Verify buffer distinction

```
__kernel void memcpy(char* tgt,
                     (char* src,
                      int length) {
  for(int i=0; i<length; i++) {
    tgt[i] = src[i];
  }
}
```

```
cl_mem srcbuf = clCreateBuffer(...);
cl_mem tgtbuf = clCreateBuffer(...);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueTask(
  queue, kernel,
  0, NULL, NULL);
```

3. Restricted Pointer Analysis

1. (Host) Identify pointed-to host buffers
2. Verify buffer distinction
3. (Device) Mark parameters restricted

```
__kernel void memcpy(char *restrict tgt,
                     (char *restrict src,
                      int length) {
  for(int i=0; i<length; i++) {
    tgt[i] = src[i];
  }
}
```

```
cl_mem srcbuf = clCreateBuffer(...);
cl_mem tgtbuf = clCreateBuffer(...);
clSetKernelArg(kernel, 0, sizeof(char*), tgtbuf);
clSetKernelArg(kernel, 1, sizeof(char*), srcbuf);
clSetKernelArg(kernel, 2, sizeof(int), length);
clEnqueueTask(
  queue, kernel,
  0, NULL, NULL);
```

Prototype Transformations

1. Geometry Propagation
2. NDRange To Loop
3. Restricted Pointer Analysis
4. **Reduction Dependence Elimination**

4. Reduction Dependency Elimination - Motivation

- Floating-point operation latency means long initiation intervals for reduction loops - the pipeline stalls on every iteration

4. Reduction Dependency Elimination - Motivation

- Floating-point operation latency means long initiation intervals for reduction loops - the pipeline stalls on every iteration
- Data dependency on the reduction variable can be resolved by using a rotating register to modulo schedule the reduction computation

4. Reduction Dependency Elimination - Motivation

- Floating-point operation latency means long initiation intervals for reduction loops - the pipeline stalls on every iteration
- Data dependency on the reduction variable can be resolved by using a rotating register to modulo schedule the reduction computation
- Pipelined reduction via a rotating register is an idiom recognized by the Intel FPGA OpenCL compiler and efficiently implemented using a shift register in hardware

4. Reduction Dependency Elimination

1. Detect reduction idiom in loops

```
__kernel void vec_sum(__global double *arr,
                      __global double *res,
                      int N)
{
  double temp_sum = 0;
  for (int i = 0; i < N; ++i)
    temp_sum += arr[i];
  *res = temp_sum;
}
```

4. Reduction Dependency Elimination

1. Detect reduction idiom in loops

```
__kernel void vec_sum(__global double *arr,
                      __global double *res,
                      int N)
{
  double temp_sum = 0;
  for (int i = 0; i < N; ++i)
    temp_sum += arr[i];
  *res = temp_sum;
}
```

4. Reduction Dependency Elimination

1. Detect reduction idiom in loops
2. Create and initialize a "shift-register" array

```
__kernel void vec_sum(__global double *arr,
                      __global double *res,
                      int N)
{
  double shift_reg[II_CYCLES + 1];
  for (int j = 0; j < II_CYCLES + 1; ++j)
    shift_reg[j] = 0;
  double temp_sum = 0;
  for (int i = 0; i < N; ++i)
    temp_sum += arr[i];
  *res = temp_sum;
}
```

4. Reduction Dependency Elimination

1. Detect reduction idiom in loops
2. Create and initialize a "shift-register" array
3. Rewrite the reduction update to store into the shift register's tail element

```
__kernel void vec_sum(__global double *arr,
                      __global double *res,
                      int N)
{
  double shift_reg[II_CYCLES + 1];
  for (int j = 0; j < II_CYCLES + 1; ++j)
    shift_reg[j] = 0;
  double temp_sum = 0;
  for (int i = 0; i < N; ++i)
    shift_reg[II_CYCLES] = shift_reg[0] + arr[i];
  *res = temp_sum;
}
```

4. Reduction Dependency Elimination

1. Detect reduction idiom in loops
2. Create and initialize a "shift-register" array
3. Rewrite the reduction update to store into the shift register's tail element
4. Shift the values of the shift register down

```
__kernel void vec_sum(__global double *arr,
                      __global double *res,
                      int N)
{
  double shift_reg[II_CYCLES + 1];
  for (int j = 0; j < II_CYCLES + 1; ++j)
    shift_reg[j] = 0;
  double temp_sum = 0;
  for (int i = 0; i < N; ++i) {
    shift_reg[II_CYCLES] = shift_reg[0] + arr[i];
    for (int k = 0; k < II_CYCLES; ++k)
      shift_reg[k] = shift_reg[k+1];
  }
  *res = temp_sum;
}
```

4. Reduction Dependency Elimination

1. Detect reduction idiom in loops
2. Create and initialize a “shift-register” array
3. Rewrite the reduction update to store into the shift register’s tail element
4. Shift the values of the shift register down
5. Compute the final reduction value by summing shift register values.

```
__kernel void vec_sum(__global double *arr,
                      __global double *res,
                      int N)
{
  double shift_reg[II_CYCLES + 1];
  for (int j = 0; j < II_CYCLES + 1; ++j)
    shift_reg[j] = 0;
  double temp_sum = 0;
  for (int i = 0; i < N; ++i) }
    shift_reg[II_CYCLES] = shift_reg[0] + arr[i];
    for (int k = 0; k < II_CYCLES; ++k)
      shift_reg[k] = shift_reg[k+1];
  }
  for (int m = 0; m < II_CYCLES; ++m)
    temp_sum += shift_reg[m];
  *res = temp_sum;
}
```

Evaluation

- OpenCL kernels taken from Rodinia benchmark suite[1]

- Execution time measured on a DE5-Net Development Kit (Stratix V)

[1] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), pp. 44-54, Oct. 2009.

Transformation Applicability

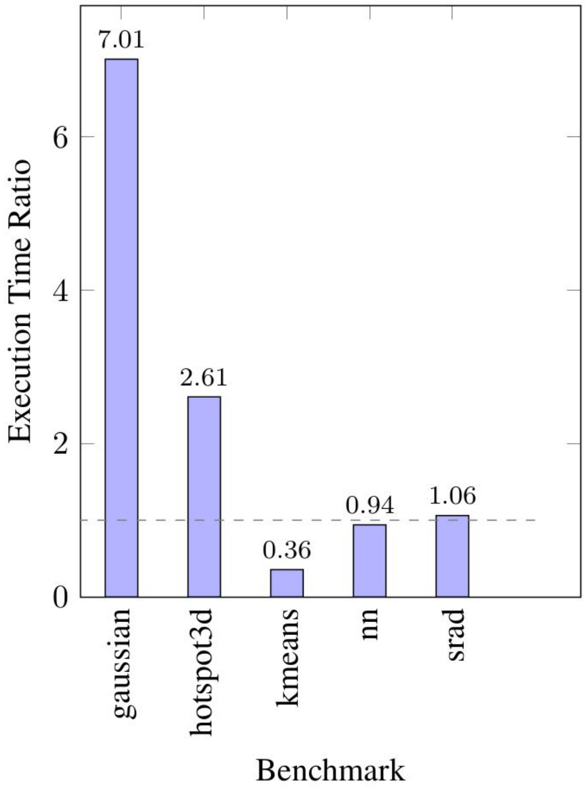
Benchmark	Restrict	NDRangeToLoop**	FloatReduce
gaussian	✘	✔	✔
hotspot3d	✔	✔	✘
kmeans	✔	✔	✘*
nn	✘	✔	✘
srad	✔	✔	✘

* An opportunity was found, but hurt performance

** Benchmarks that could not be transformed by **NDRangeToLoop** were excluded from evaluation

Results

Gaussian	**7x slower**
Hotspot3D	**2.6x slower**
Kmeans	**2.8x faster**
NN	**6% faster**
SRAD	**6% slower**



What Happened?

Analysis

1. Inter-compiler Communication
2. Compiler Versions
3. Missing Heuristics

Inter-Compiler Communication

- Loops created by NDRangeToLoop carry no dependencies,
  but the Intel FPGA OpenCL Compiler doesn’t read that information

Inter-Compiler Communication

- Loops created by NDRangeToLoop are carry no dependencies,
  but the Intel FPGA OpenCL Compiler doesn’t read that information

- Often, the Intel FPGA OpenCL Compiler cannot rediscover the parallelism,
  And cannot pipeline

Compiler Versions

- The Intel FPGA OpenCL Compiler is built on LLVM 3.0 (circa 2011)

Compiler Versions

- The Intel FPGA OpenCL Compiler is built on LLVM 3.0 (circa 2011)

- Modern LLVM Analyses & Transformations ineffective or unavailable:
  AssumptionCache
  Loop Vectorization
  Interprocedural Alias Analysis

Missing Heuristics

- Our prototype cannot access Intel FPGA OpenCL compiler's heuristics, only IR between stages

Missing Heuristics

- Our prototype cannot access Intel FPGA OpenCL Compiler's heuristics, only IR between stages

- Our transformations **do not** know if they're helping or hurting

Missing Heuristics

- Our prototype cannot access Intel FPGA OpenCL Compiler's heuristics, only IR between stages

- Our transformations **do not** know if they're helping or hurting

- An open-source compiler would help a lot with this

Conclusion

Compilers can perform much more powerful transformations when able to inspect and affect both host and device compilation.

Compilers can perform much more powerful transformations when able to inspect and affect both host and device compilation.

Deep integration is required to determine if transformations improve performance.

Contact Me

Taylor Lloyd - tjlloyd@ualberta.ca