GPU Programming in Rust
Implementing High-level Abstractions in a Systems-level Language

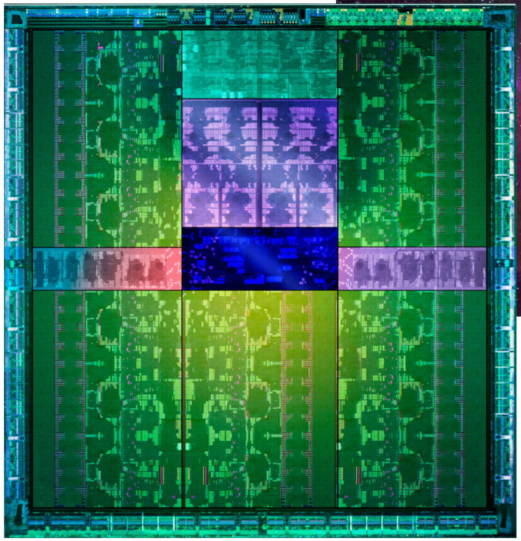
Eric Holk, Milinda Pathirage, Arun Chauhan, Andrew Lumsdaine
Indiana University

Nicholas D. Matsakis
Mozilla Research

May 20, 2013

CENTER FOR RESEARCH
IN EXTREME SCALE
TECHNOLOGIES
INDIANA UNIVERSITY
Pervasive Technology Institute

Motivation



Motivation

Two approaches to GPU programming

Motivation

Two approaches to GPU programming

Systems-level

```
__kernel void dotprod(__global float *x, __global float *y,
                      __global float *z, int N)
{
    size_t num_groups = get_num_groups(0);
    int block_size = (N + num_groups - 1) / num_groups;
    int block_id = get_global_id(0) / get_local_size(0);

    int block_start = block_id * block_size;
    int block_end = min(block_start + block_size, N);

    __local float temp[LOCAL_SIZE];

    int i = get_local_id(0);

    // Phase 1: reduce down to a size that fits in local memory.
    float t = 0;
    for(int j = block_start + i; j < block_end; j += LOCAL_SIZE) {
        t += x[j] * y[j];
    }
    temp[i] = t;

    // Phase 2: sum up the temporary array
    for(int j = LOCAL_SIZE / 2; j >= CUTOFF; j >>= 1) {
        barrier(CLK_LOCAL_MEM_FENCE);
        if(i < j) {
            temp[i] += temp[i + j];
        }
    }

    if(i < CUTOFF) {
        z[block_id * CUTOFF + i] = temp[0];
    }
}
```

```
__kernel void dotprod(__global float *x, __global float *y,
                      __global float *z, int N)
{
    size_t num_groups = get_num_groups(0);
    int block_size = (N + num_groups - 1) / num_groups;
    int block_id = get_global_id(0) / get_local_size(0);

    int block_start = block_id * block_size;
    int block_end = min(block_start + block_size, N);

    __local float temp[LOCAL_SIZE];

    int i = get_local_id(0);

    // Phase 1: reduce down to a size that fits in local memory.
    float t = 0;
    for(int j = block_start + i; j < block_end; j += LOCAL_SIZE) {
        t += x[j] * y[j];
    }
    temp[i] = t;

    // Phase 2: sum up the temporary array
    for(int j = LOCAL_SIZE / 2; j >= CUTOFF; j >>= 1) {
        barrier(CLK_LOCAL_MEM_FENCE);
        if(i < j) {
            temp[i] += temp[i + j];
        }
    }

    if(i < CUTOFF) {
        z[block_id * CUTOFF + i] = temp[0];
    }
}
```

Explicit memory placement

```
__kernel void dotprod(__global float *x, __global float *y,
                      __global float *z, int N)
{
    size_t num_groups = get_num_groups(0);
    int block_size = (N + num_groups - 1) / num_groups;
    int block_id = get_global_id(0) / get_local_size(0);

    int block_start = block_id * block_size;
    int block_end = min(block_start + block_size, N);

    __local float temp[LOCAL_SIZE];

    int i = get_local_id(0);

    // Phase 1: reduce down to a size that fits in local memory.
    float t = 0;
    for(int j = block_start + i; j < block_end; j += LOCAL_SIZE) {
        t += x[j] * y[j];
    }
    temp[i] = t;

    // Phase 2: sum up the temporary array
    for(int j = LOCAL_SIZE / 2; j >= CUTOFF; j >>= 1) {
        barrier(CLK_LOCAL_MEM_FENCE);
        if(i < j) {
            temp[i] += temp[i + j];
        }
    }

    if(i < CUTOFF) {
        z[block_id * CUTOFF + i] = temp[0];
    }
}
```

Aware of computation structure

```
__kernel void dotprod(__global float *x, __global float *y,
                      __global float *z, int N)
{
    size_t num_groups = get_num_groups(0);
    int block_size = (N + num_groups - 1) / num_groups;
    int block_id = get_global_id(0) / get_local_size(0);

    int block_start = block_id * block_size;
    int block_end = min(block_start + block_size, N);

    __local float temp[LOCAL_SIZE];

    int i = get_local_id(0);

    // Phase 1: reduce down to a size that fits in local memory.
    float t = 0;
    for(int j = block_start + i; j < block_end; j += LOCAL_SIZE) {
        t += x[j] * y[j];
    }
    temp[i] = t;

    // Phase 2: sum up the temporary array
    for(int j = LOCAL_SIZE / 2; j >= CUTOFF; j >>= 1) {
        barrier(CLK_LOCAL_MEM_FENCE);
        if(i < j) {
            temp[i] += temp[i + j];
        }
    }

    if(i < CUTOFF) {
        z[block_id * CUTOFF + i] = temp[0];
    }
}
```

Explicit synchronization

```
__kernel void dotprod(__global float *x, __global float *y,
                      __global float *z, int N)
{
    size_t num_groups = get_num_groups(0);
    int block_size = (N + num_groups - 1) / num_groups;
    int block_id = get_global_id(0) / get_local_size(0);

    int block_start = block_id * block_size;
    int block_end = min(block_start + block_size, N);

    __local float temp[LOCAL_SIZE];

    int i = get_local_id(0);

    // Phase 1: reduce down to a size that fits in local memory.
    float t = 0;
    for(int j = block_start + i; j < block_end; j += LOCAL_SIZE) {
        t += x[j] * y[j];
    }
    temp[i] = t;

    // Phase 2: sum up the temporary array
    for(int j = LOCAL_SIZE / 2; j >= CUTOFF; j >>= 1) {
        barrier(CLK_LOCAL_MEM_FENCE);
        if(i < j) {
            temp[i] += temp[i + j];
        }
    }

    if(i < CUTOFF) {
        z[block_id * CUTOFF + i] = temp[0];
    }
}
```

Motivation

Two approaches to GPU programming

Motivation

Two approaches to GPU programming

High-level

```
@cu
def dot_product(x, y):
    return sum(map(op_mul, x, y))
```

```
@cu
def dot_product(x, y):
    return sum(map(op_mul, x, y))
```

Black boxes

Motivation

Can we have both approaches at once?

Approach

Add support for GPU kernels in Rust, using LLVM PTX target

Use Rust language features to build higher level abstractions

Rust

New systems language from Mozilla Research

Rust

Concurrency-aware memory model

Rust

Concurrency-aware memory model

```
let shared_vec = @[1, 2, 3, 4];

let owned_vec  = ~[1, 2, 3, 4];
```

Rust

Concurrency-aware memory model

```
let shared_vec = @[1, 2, 3, 4];

let owned_vec  = ~[1, 2, 3, 4];
```

Garbage collection is optional

Rust

Zero-cost abstraction

Rust

Zero-cost abstraction

Polymorphism by monomorphization

Relies heavily on LLVM optimizations

Rust

Library-based iteration idioms

```
fn range(start: int, stop: int, f: fn(int) -> bool) { ... }
```

Rust

Library-based iteration idioms

```
fn range(start: int, stop: int, f: fn(int) -> bool) { ... }
```

```
for range(0, 100) |i| {
    print(fmt!("%d\n", i));
}
```

GPU Support in Rust

GPU Support in Rust

Accomplished by LLVM's PTX code generation target

GPU Support in Rust

Accomplished by LLVM's PTX code generation target

#[kernel] annotation indicates code that runs on GPU

```
#[kernel]
fn add_float(x: &float,
             y: &float,
             z: &mut float)
{
  *z = *x + *y
}
```

GPU Support in Rust

Accomplished by LLVM's PTX code generation target

#[kernel] annotation indicates code that runs on GPU

Thread ID, etc. are exposed through intrinsic functions

```
#[kernel]
fn add_vectors(++x: ~[float],
               ++y: ~[float],
               ++z: ~[mut float])
{
  let i = ptx_ctaid_x() * ptx_ntid_x() + ptx_tid_x();
  z[i] = x[i] + y[i];
}
```

Building abstractions

Follow Rust idioms when possible

Building abstractions

Mapping

```
#[kernel]
fn add_vectors(  N: uint,
               ++A: ~[float],
               ++B: ~[float],
               ++C: ~[mut float])
{
  do gpu::range(0, N) |i| {
      C[i] = A[i] + B[i]
  };
}
```

Building abstractions

Reduction

```
fn reduce_into<T>(target: &mut T,
                  init: T,
                  source: &[const T],
                  op: fn&(T, T) -> T);
```

```
#[kernel]
fn vector_sum(++src: ~[float],
              dst: &mut float)
{
    gpu::reduce_into(dst, 0f, src,
                     |a, b| a + b);
}
```

Building abstractions

Stencils

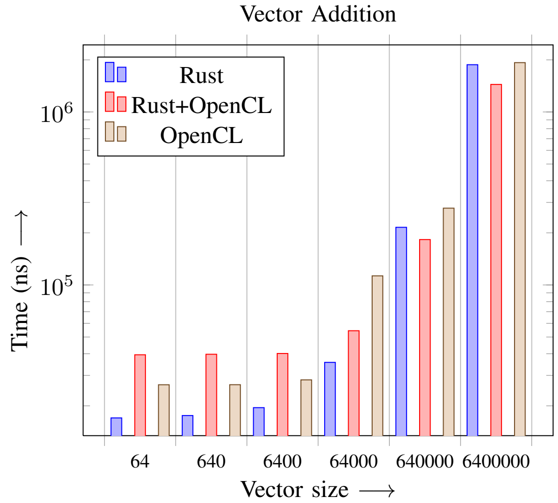
```
fn stencil_into_5pt<T>(
  shape: (uint, uint),
  target: &[mut T],
  source: &[const T],
  op: fn&(T, T, T, T, T) -> T);


#[kernel]
fn jacobi(++src: ~[float],
                dst: &mut float)
{
    do gpu::stencil_into_5pt((N, N),
                             src,
                             dst)
    |   u,
     l, c, r,
        d    | {

     (u + l + r + d) / 4f
    }
}
```

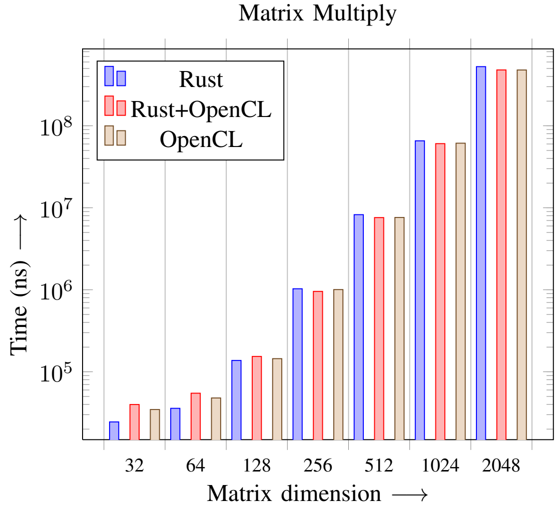
Building abstractions

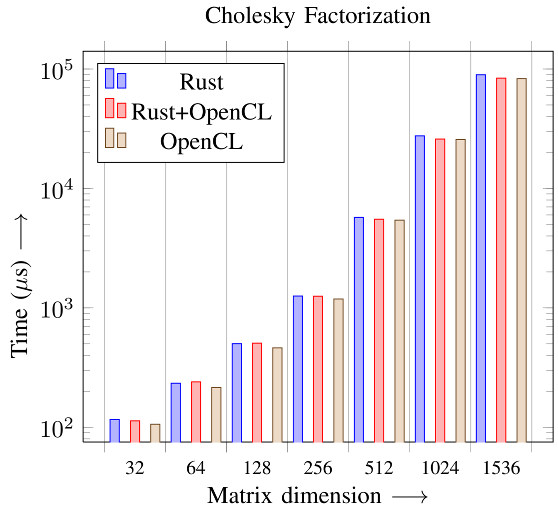
Rust has a rich set of language features to build higher level abstractions for GPUs.

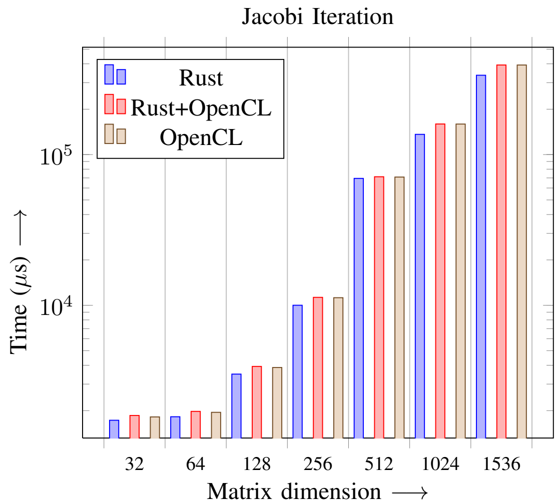
Performance

Performance – Kernel Execution Time









Summary

Summary

LLVM simplifies adding low-level GPU support to existing languages.

Summary

LLVM simplifies adding low-level GPU support to existing languages.

Language features can be used to build high level support for GPU programming.

Thanks!

Eric Holk
`eholk@cs.indiana.edu`

`http://www.rust-lang.org/`

`https://github.com/eholk/RustGPU`

Performance – Memory Transfer Time