BUILDING PARALLEL ALGORITHMS WITH BULK

Jared Hoberock
Programming Systems and Applications
NVIDIA Research
github.com/jaredhoberock
#include <cstdio>
#include <bulk/bulk.hpp>

struct hello
{
  __device__
  void operator()()
  {
    printf("Hello, brother!\n");
  }

  __device__
  void operator()(bulk::parallel_group<> &g)
  {
    printf("Hello, brother from Bulkamaniac %d\n", g.this_exec.index());
  }
};

int main()
{
  bulk::async(bulk::par(1), hello());
  bulk::async(bulk::par(32), hello(), bulk::root).wait();
  return 0;
}
BULK
Want a good way to author GPU kernels

- Maintainability
- Robustness
- Composability
- Productivity
- Performance
MAINTAINABILITY

● When a Thrust unit test fails it’s WrestleMania

● Want small functions testable in isolation
  ○ well-defined preconditions
  ○ well-defined postconditions

● Should be able to tell at a glance what code does 18 months later
ROBUST

● Shouldn’t require 1000 lines of code to launch a kernel

● Bulk smooths over rough edges
  ○ CUDA: you asked for too many blocks!
  ○ CUDA: your parameters are too large!
  ○ CUDA: your parameters aren’t POD!
  ○ CUDA: you asked for too much shared memory!
  ○ CUDA: you don’t have a GPU!

● Just make it work
COMPOSABILITY

● __shared__ memory
  ○ Current practice requires meticulous allocation
  ○ Count up (somehow) requirements in advance
  ○ Suballocate, reinterpret

● Should work more like malloc

● Virtualization
void f1(int *array1)
{
    // use array1...
}

void f2(double *array2)
{
    // use array2...
}

void f3()
{
    union shared
    {
        int array1[BLOCK_SIZE];
        double array2[BLOCK_SIZE];
    }

    __shared__ shared smem;

    f1(smem.array1);
    f2(smem.array2);
}
PRODUCTIVITY

- Groups of Execution Agents
- Example: SAXPY
- Example: sum
- Example: merge
EXECUTION AGENTS

- Not necessarily physical threads
- Logical lightweight threads
- Parameterized by grainsize
  - quantum of sequential work
- Grouped in a tree structure
```cpp
#include <iostream>
#include <thrust/device_vector.h>
#include <bulk/bulk.hpp>

struct saxpy
{
  __device__
  void operator()(bulk::agent<> &self, float a, float *x, float *y)
  {
    int i = self.index();
    y[i] = a * x[i] + y[i];
  }
};

int main()
{
  size_t n = 1 << 24;
  thrust::device_vector<float> x(n,1);
  thrust::device_vector<float> y(n,1);

  bulk::async(bulk::par(n), saxpy(), bulk::root.this_exec,
              13, thrust::raw_pointer_cast(x.data()), thrust::raw_pointer_cast(y.data()));

  return 0;
}
```
LOOK, MA!

bulk::async(bulk::par(n), saxpy(), ...);

- No launch config
- No decomposition
- No parameter marshalling
- Easy stuff is easy
AUTOMATIC LAUNCH CONFIG

- `bulk::par(n)` automatically configures kernel launch
- Occupancy-promoting heuristic
- Largest block size which maximizes occupancy
bulk::async

template<typename ExecutionGroup, typename Function, typename... Args>
future<void> async(ExecutionGroup &&exec, Function f, Args... &&args);

- Asynchronous kernel launcher
- `ExecutionGroup` describes tree
- If any argument is a cursor, performs substitution
  - opts-in to virtualization
  - otherwise, just refer to `threadIdx` & friends
  - avoids a walled garden
**bulk::parallel_group**

- Instantiated by `bulk::async`
- Handle to the current logical thread block
- Knows its size
  - Size is dynamic or static
    - `bulk::parallel_group<>`
    - `bulk::parallel_group<bulk::agent<>, 128>`
  - Static sizing important optimization
bulk::concurrent_group

- Adds concurrency requirement to parallel_group
- Has a barrier: `g.wait()`;
- Allows structured cooperation/communication
- Can’t really be virtualized
#include <cassert>
#include <thrust/device_vector.h>
#include <bulk/bulk.hpp>

struct sum { ... };

int main()
{
  size_t n = 512;
  thrust::device_vector<int> vec(n, 1);
  thrust::device_vector<int> result(1);

  bulk::async(bulk::con(n), sum(), bulk::root, vec.data(), result.data());

  assert(512 == result[0]);
}
struct sum
{
    __device__
    void operator()(bulk::concurrent_group<> &g, 
                   thrust::device_ptr<int> data, 
                   thrust::device_ptr<int> result)
    {
        int s = bulk::reduce(g, data, data + g.size());

        if(g.this_exec.index() == 0)
        {
            *result = s;
        }
    }
};
CONCURRENT ALGORITHMS

template<typename ConcurrentGroup, typename Iterator, typename T>
T reduce(ConcurrentGroup &g, Iterator first, Iterator last, T init);

● Execution group passed as first parameter
  ○ cf. C++17 parallel execution policies

● Invoked convergently

● Invocations synchronize
template<typename ConcurrentGroup, typename Iterator, typename T>
__device__
T reduce(ConcurrentGroup &g, Iterator first, Iterator last, T init)
{
    unsigned int n = last - first;

    T *s_data = (T*)bulk::malloc(g, n * sizeof(T));

    bulk::copy_n(g, data, n, s_data);

    while(n > 1)
    {
        unsigned int half_n = n / 2;

        if(g.this_exec.index() < half_n)
        {
            s_data[g.this_exec.index()] += s_data[n - g.this_exec.index() - 1];
        }

        g.wait();

        n -= half_n;
    }

    T result = s_data[0];

    g.wait();

    bulk::free(g, s_data);

    return result;
}
CONCURRENT ALGORITHMS

accumulate  reduce
copy          reduce_by_key
for_each      inclusive_scan
gather        exclusive_scan
merge          scatter
sort           adjacent_difference
SHARED HEAP

- `bulk::async` automatically allocates `__shared__` memory
- Occupancy-based heuristic
- Programmer can explicitly request a heap size
  - backed by on-chip, then DRAM
  - graceful fallback
void f1(int *array1)
{
    // use array1...
}

void f2(double *array2)
{
    // use array2...
}

void f3()
{
    union shared
    {
        int array1[BLOCK_SIZE];
        double array2[BLOCK_SIZE];
    }

    __shared__ shared smem;

    f1(smem.array1);
    f2(smem.array2);
}
void f1(bulk::concurrent_group<> &g)
{
    int sz = sizeof(int) * g.size();

    int *array1 = (int*)bulk::malloc(g, sz);

    // use array1...
    bulk::free(g, array1);
}

void f2(bulk::concurrent_group<> &g)
{
    int sz = sizeof(double) * g.size();

    double *array2 = (double*)bulk::malloc(g, sz);

    // use array2...
    bulk::free(g, array2);
}

void f3(bulk::concurrent_group<> &g)
{
    f1(g);
    f2(g);
}
GROUPS IN PARALLEL

```c
#include <thrust/device_vector.h>
#include <bulk/bulk.hpp>

struct reduce_intervals { ... };

int main()
{
    size_t num_intervals = 4;
    size_t interval_size = 256;

    thrust::device_vector<int> input(num_intervals * interval_size, 1);
    thrust::device_vector<int> sums(num_intervals);

    // launch some concurrent_groups in parallel
    bulk::async(bulk::par(bulk::con(interval_size), num_intervals),
               reduce_intervals(), bulk::root, input.data(), interval_size, sums.data());

    return 0;
}
```
GROUPS IN PARALLEL

```cpp
struct reduce_intervals
{
    __device__
    void operator()(bulk::parallel_group<bulk::concurrent_group<> > &g,
                   thrust::device_ptr<int> input,
                   size_t interval_size,
                   thrust::device_ptr<int> sums)
    {
        // get the group's index
        int idx = g.this_exec.index();
        size_t offset = idx * interval_size;

        // reduce this group's interval of data
        int sum = bulk::reduce(g.this_exec, input + offset, input + offset + interval_size, 0);

        // the first agent stores the sum
        if(g.this_exec.this_exec.index() == 0)
        {
            sums[idx] = sum;
        }
    }
};
```
GROUPS IN PARALLEL

```cpp
#include <thrust/device_vector.h>
#include <bulk/bulk.hpp>

struct reduce_intervals { ... };

int main()
{
    size_t num_intervals = 4;
    size_t interval_size = 256;

    thrust::device_vector<int> vec(num_intervals * interval_size, 1);
    thrust::device_vector<int> sums(num_intervals);

    bulk::async(bulk::grid(num_intervals, interval_size),
                reduce_intervals(), bulk::root, vec.data(), interval_size, sums.data());

    return 0;
}
```
EXECUTION HIERARCHY

- Execution groups nest
  - `bulk::parallel_group<
    - `bulk::concurrent_group<
      - `bulk::agent<>`

- Corresponds to CUDA grid
- Only a few nestings implemented
- Traverse down the hierarchy
  - `group.this_exec.this_exec...`
INPLACE MERGE

#include <thrust/device_vector.h>
#include <thrust/tabulate.h>
#include <bulk/bulk.hpp>

struct merge_kernel { ... };

void merge(int *a, int n1, int *b, int n2, int *c)
{
    const int groupsize = 256;
    const int grainsize = 9;
    const int tile_size = groupsize * grainsize;

    int num_groups = (n1 + n2 + tile_size - 1) / tile_size;

    thrust::device_vector<int> merge_paths(num_groups + 1);

    thrust::tabulate(merge_paths.begin(), merge_paths.end(),
                     locate_merge_path(tile_size, a, n1, b, n2));

    // merge partitions
    bulk::async(bulk::grid<groupsize, grainsize>(num_groups),
                merge_kernel(), bulk::root.this_exec, a, n1, b, n2, c)
}
struct merge_kernel
{
    template<std::size_t groupsize, std::size_t grainsize>
    __device__
    void operator()(bulk::concurrent_group<bulk::agent<grainsize>, groupsize> &g,
                   int *a, int n1, int *b, int n2, int *merge_paths, int *c)
    {
        // do some index bookkeeping to find this group's data
        int mp0 = merge_paths[g.index()];
        int mp1 = merge_paths[g.index() + 1];
        int tile_size = groupsize * grainsize;
        int diag = tile_size * g.index();

        int local_size1 = mp1 - mp0;
        int local_size2 = min(n1 + n2, diag + tile_size) - mp1 - diag + mp0;

        // point to our group's input & output
        a += mp0;
        b += diag - mp0;
        result += tile_size * g.index();

        // allocate a staging buffer
        int *stage = (int*)bulk::malloc(g, tile_size * sizeof(int));
        staged_merge(g, a, local_size1, b, local_size2, stage, result);
        bulk::free(g, stage);
    }
};
template<std::size_t groupsize, std::size_t grainsize>
__device__
void staged_merge(bulk::concurrent_group<bulk::agent<grainsize>,groupsize> &g,
                      int *a, int n1, int *b, int n2, int *stage, int *c)
{
    // copy into the stage
    bulk::copy_n(bulk::bound<groupsize * grainsize>(g),
                   make_join_iterator(a, n1, b),
                   n1 + n2,
                   stage);

    // inplace merge in the stage
    bulk::inplace_merge(bulk::bound<groupsize * grainsize>(g),
                         stage, stage + n1, stage + n1 + n2);

    // copy to the result
    bulk::copy_n(bulk::bound<groupsize * grainsize>(g),
                   stage, n1 + n2, c);
}
template<std::size_t groupsize, std::size_t grainsize>
__device__
void staged_merge(bulk::concurrent_group<bulk::agent<grainsize>,groupszie> &g,
   int *a, int n1, int *b, int n2, int *stage, int *c)
{
   // copy into the stage
   bulk::copy_n(bulk::bound<groupszie * grainsize>(g),
      make_join_iterator(a, n1, b),
      n1 + n2,
      stage);

   // inplace merge in the stage
   bulk::inplace_merge(bulk::bound<groupszie * grainsize>(g),
      stage, stage + n1, stage + n1 + n2);

   // copy to the result
   bulk::copy_n(bulk::bound<groupszie * grainsize>(g),
      stage, n1 + n2, c);
}
**BOUNDING EXECUTION**

- `bulk::bound<b>(g)` provides an upper bound on the size of an algorithm's result

- Important optimization

- Enables a single algorithm interface
template<\size\_bound, \size\_groupsize, \size\_grainsize>
 DEVICE
 typename enable_if<
 bound <= groupsize * grainsize
 >::type
 inplace_merge(bulk::bounded<
 bound,
 bulk::concurrent_group<
 bulk::agent<grainsize>,
 groupsize
 >
 > &g,
 int *first, int *middle, int *last)
 {
 int n1 = middle - first;
 int n2 = last - middle;

 int local_offset = grainsize * g.this_exec.index();
 int mp = bulk::merge_path(first, n1, middle, n2, local_offset);

 int local_offset1 = mp;
 int local_offset2 = n1 + local_offset - mp;

 int local_result[grainsize];

 // merge sequentially with each agent
 bulk::merge(bulk::bound<grainsize>(g.this_exec),
 first + local_offset1, middle,
 first + local_offset2, last,
 local_result);
 g.wait();

 int local_size = max(0, min(grainsize, n1 + n2 - local_offset));

 bulk::copy_n(bulk::bound<grainsize>(g.this_exec), local_result, local_size, first + local_offset);
 g.wait();
 }
MORE EXAMPLES ON GITHUB

hello_world
async_reduce
for_each
futures
merge
merge_sort_by_key

ping_pong
reduce
reduce_by_key
saxpy
scan
sum
<table>
<thead>
<tr>
<th>Function</th>
<th>Thrust LOC</th>
<th>Bulk LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>for_each</td>
<td>199</td>
<td>10</td>
</tr>
<tr>
<td>reduce</td>
<td>275</td>
<td>85</td>
</tr>
<tr>
<td>inclusive_scan</td>
<td>753</td>
<td>134</td>
</tr>
<tr>
<td>merge</td>
<td>285</td>
<td>150</td>
</tr>
<tr>
<td>reduce_by_key</td>
<td>705</td>
<td>255</td>
</tr>
</tbody>
</table>

Before vs After
PERFORMANCE: REDUCE
PERFORMANCE: REDUCE
PERFORMANCE: SCAN

![Graph showing 32b int Scan Bandwidth (K20c)]
PERFORMANCE: MERGE

32b int Merge Bandwidth (K20c)

- MGPU
- Bulk (no heap)
- Bulk (implicit heap size)
- Bulk (explicit heap size)
- Thrust

64b float Merge Bandwidth (K20c)

- Bulk (no heap)
- MGPU
- Bulk (implicit heap size)
- Bulk (explicit heap size)
- Thrust

GBs vs Millions of elements
PERFORMANCE: REDUCE_BY_KEY

32b int Reduce by Key Bandwidth (K20c)

32b float Reduce by Key Bandwidth (K20c)
We don’t lose much when targeting a higher level of abstraction
FUTURE WORK

- Simpler, faster
- Expose in Thrust?
- CPUs
- Distributed environments
- Ultimate goal: truly performance portable codes
github.com/jaredhoberock/bulk
github.com/jaredhoberock/shmalloc