Programming Massively Parallel Hardware with Agency

Jared Hoberock & Michael Garland
NVIDIA Research Programming Systems and Applications Group
October 2016

NVIDIA

RESEARCH QUESTION

How might programmers write highly parallel programs in a mainstream language like C++?

LOOKING AHEAD

Agency: a lower-level framework for parallel execution

```
void saxpy(size_t n, float a, const float* x, const float* y, float* z)
{
  using namespace agency;

  bulk_invoke(par(n), [=](parallel_agent &self)
  {
    int i = self.index();
    z[i] = a * x[i] + y[i];
  });
}
```

BEGIN AT THE BEGINNING

Independent loop iterations represent latent parallelism

```
void saxpy(int n, float a, float *x, float *y)
{
   // Sequential code with latent parallelism
   for(int i=0; i<n; ++i)
   {
        y[i] = a*x[i] + y[i];
   }
}
```

PARALLEL LOOPS IN C++17

Library implementation of parallel constructs

```
void saxpy(int n, float a, float *x, float *y)
{
  auto I = interval(0, n);

  std::for_each(std::execution::par, I.begin(), I.end(), [=](int i)
  {
      y[i] = a*x[i] + y[i];
  }
}
```

PARALLEL LOOPS

Increasingly common in standard languages

OpenMP	`#pragma omp parallel for` `for(int i=a; i<b; ++i) { ... }`
OpenACC	`#pragma acc loop` `for(int i=a; i<b; ++i) { ... }`
Fortran 2008	`DO CONCURRENT (I=1:N) ... END DO`
C++17	`std::for_each(std::execution::par, begin, end, [](int i) { ... });`

STANDARD TEMPLATE LIBRARY

Higher-level library built around algorithms

```
for_each(begin, end, function);
```

Operator	Data	Function
A named pattern of computation and communication.	One or more collections to operate on.	Caller-provided function object injected in pattern.

C++17 PARALLEL STL

Algorithms + Execution Policies

```
for_each(par, begin, end, function);
```

Execution Policy

Specify *how* operation may execute.

C++ Extensions for Parallelism

Draft technical specification:
http://wg21.link/n4507

EXECUTION POLICIES

Specify how algorithms may execute

POLICY NAME	MEANING
`seq`	Sequential execution alone is permitted.
`par`	Parallel execution is permitted.
`par_unseq`	Vectorized parallel execution is permitted.

parallel :: provided function objects can be executed in any order on one or more threads.

vectorized :: provided function objects can be also be interleaved when on one thread.

PARALLEL ALGORITHMS

Many useful patterns beyond loops

Parallelizable algorithms in STL

`for_each`

`transform`

`copy_if`

`sort`

`set_intersection`

etc.

New additions for parallelism

`reduce`

`exclusive_scan`

`inclusive_scan`

`transform_reduce`

`transform_inclusive_scan`

`transform_exclusive_scan`

LARGE VOCABULARY OF ALGORITHMS

How might parallel programmers implement these algorithms?

AGENCY FRAMEWORK

Developing a lower-level framework for parallel execution

A C++ template library for abstracting execution

Central primitives are execution agents

Portable, with efficient mapping to underlying platform

Extensibility via executors

AGENCY FRAMEWORK

Developing a lower-level framework for parallel execution

```
void saxpy(size_t n, float a, const float* x, const float* y, float* z)
{
  using namespace agency;

  bulk_invoke(par(n), [=](parallel_agent &self)
  {
    int i = self.index();
    z[i] = a * x[i] + y[i];
  });
}
```

AGENCY FRAMEWORK

Developing a lower-level framework for parallel execution

```
void saxpy(size_t n, float a, const float* x, const float* y, float* z)
{
  using namespace agency;

  auto executor = get_desired_executor();

  bulk_invoke(par(n).on(executor), [=](parallel_agent &self)
  {
    int i = self.index();
    z[i] = a * x[i] + y[i];
  });
}
```

AGENCY’S EXECUTION POLICIES

Specify how control structures execute work

POLICY NAME	MEANING
`seq`	Sequential execution alone is permitted.
`par`	Parallel execution is permitted.
`par_unseq`	Vectorized parallel execution is permitted.
`unseq`	Vectorized execution is permitted in the current thread.
`con`	Concurrent forward progress is mandatory.

IMPLEMENTING ALGORITHMS

Requires suitable lower level constructs

```
template<class Range>
int sum(Range&& data)
{
    // organize data into partitions
    auto partitions = make_partitioned_view(data, par.executor().shape());

    // sum each partition
    auto partial_sums = bulk_invoke(par, [=](auto& self)
    {
        return sum(seq, partitions[self.index()]);
    });

    // reduce the partial sums
    return sum(seq, partial_sums);
}
```

IMPLEMENTING ALGORITHMS

But waiting can be harmful

```
template<class Range>
int sum(Range&& data)
{
    // organize data into partitions
    auto partitions = make_partitioned_view(data, par.executor().shape());

    // sum each partition
    auto partial_sums = bulk_invoke(par, [=](auto& self)
    {
        return sum(seq, partitions[self.index()]);
    });
                                                                          wait
    // reduce the partial sums
    return sum(seq, partial_sums);
                                                                          wait
}
```

ADDING ASYNCHRONY

Future-based machinery

```
template<class Range>
future<int> async_sum(Range&& data)
{
    // organize data into tiles
    auto tiles = tile_evenly(data, par.executor().unit_shape());

    // sum each tile
    auto partial_sums = bulk_async(par, [=](auto& self)
    {
        return sum(seq, tiles[self.index()]);
    });

    // reduce the partial sums
    return partial_sums.then([=](auto& partial_sums)
    {
        sum(seq, partial_sums);
    });
}
```

ADDING ASYNCHRONY

But phasing can be inefficient

```
template<class Range>
future<int> async_sum(Range&& data)
{
    // organize data into tiles
    auto tiles = tile_evenly(data, par.executor().unit_shape());

    // sum each tile
    auto partial_sums = bulk_async(par, [=](auto& self)
    {
        return sum(seq, tiles[self.index()]);
    });

    // reduce the partial sums
    return partial_sums.then([=](auto& partial_sums)
    {
        sum(seq, partial_sums);
    });
}
```

phase 1

phase 2

ADDING CONCURRENCY

Flexible synchronization

```
template<class Range>
future<int> async_sum(Range&& data)
{
    // organize data into tiles
    auto tiles = tile_evenly(data, con.executor().unit_shape());

    // sum each tile
    return bulk_async(con, [=](auto& self, vector<int>& partials) -> single_result<int>
    {
        partials[self.index()] = sum(seq, tiles[self.index()]);

        self.wait();

        // agent 0 computes the final sum
        if(self.index() == 0) return sum(seq, partials);
        else return ignore;
    },
    share<vector<int>>(partitions.size()));
}
```

WHERE DOES EXECUTION ACTUALLY HAPPEN?

Diverse
Control
Structures

`bulk_async(...)` `for_each(...)`

`sort(...)` `bulk_invoke(...)`

`your_favorite_control_structure(...)`

Multiplicative
Explosion

×

Diverse
Execution
Resources

Operating System Threads

Thread pool schedulers

OpenMP runtime

SIMD vector units

GPU runtime

Fibers

Diverse Control Structures

`bulk_async(...)` `for_each(...)`

`sort(...)` `bulk_invoke(...)`

`your_favorite_control_structure(...)`

Uniform Abstraction

Executors

Diverse Execution Resources

Operating System Threads

Thread pool schedulers

OpenMP runtime

SIMD vector units

GPU runtime

Fibers

EXECUTOR FRAMEWORK

Abstract platform details of execution

Create execution agents

Manage data they share

Advertise semantics

Mediate dependencies

```
class sample_executor
{
public:
  using execution_category = ...;

  using shape_type = tuple<size_t,size_t>;

  template<class T>
  using future = ...;

  template<class T>
  future<T> make_ready_future(T&& value);

  template<class Function, class Factory1, class Factory2>
  future<...> bulk_async_execute(Function f,
                                 shape_type shape,
                                 Factory1 result_factory
                                 Factory2 shared_factory);
  ...
};
```

See http://wg21.link/p0058r1 for details.

PURPOSE OF EXECUTORS

Provide control over where/how execution happens

Placement is, by default, at discretion of the system.

```
for_each(par, I.begin(), I.end(), [](int i) { y[i] += a*x[i]; });
```

In some cases, the programmer might want to control placement.

```
auto exec1 = choose_some_executor();
auto exec2 = choose_another_executor();

for_each(par.on(exec1), I.begin(), I.end(), ...);
for_each(par.on(exec2), I.begin(), I.end(), ...);
```

PURPOSE OF EXECUTORS

Control relationship with calling thread

```
async(launch_flags, function);
```

```
async(executor, function);
```

EXECUTOR INTERFACE

Semantic types exposed by executors

TYPE	MEANING
`execution_category`	Scheduling semantics amongst agents in a task. (sequenced, vector-parallel, parallel, concurrent)
`shape_type`	Type for indexing bulk launch of agents. (typically n-dimensional integer indices)
`future<T>`	Type for synchronizing asynchronous activities. (follows interface of `std::future`)

EXECUTOR INTERFACE

Three core constructs for launching work

Single-agent Tasks

```
result sync_execute(Function f);

future<result> async_execute(Function f);

future<result> then_execute(Function f, Future& predecessor);
```

Multi-agent Tasks

```
result bulk_sync_execute(Function f,
                         shape_type shape, Factory result_factory, Factory shared_factory);

future<result> bulk_async_execute(Function f,
                                  shape_type shape, Factory result_factory, Factory shared_factory);

future<result> bulk_then_execute(Function f, Future& predecessor,
                                 shape_type shape, Factory result_factory, Factory shared_factory);
```

SIMPLE EXAMPLE

An executor targeting OpenMP

```
struct omp_executor
{
  using execution_category = parallel_execution_tag;

  template<class Function, class R, class S>
  auto bulk_sync_execute(Function f, size_t n, R result_f, S shared_f)
  {
    auto result = result_f();
    auto shared_arg = shared_f();

    #pragma omp parallel for
    for(size_t i = 0; i < n; ++i)
    {
      f(i, result, shared_arg);
    }

    return result;
  }
};
```

EXECUTOR COLLECTIONS

Present a collection of executors as a single logical executor

```
my_executor ex0, ex1, ex2;

executor_array<my_executor> exec_array{ex0, ex1, ex2};

exec_array.bulk_sync_execute([](auto idx)
{
   // work on task (i, j)
   ...
},
{3, 2} // create nested agent groups
);
```

EXECUTOR COLLECTIONS

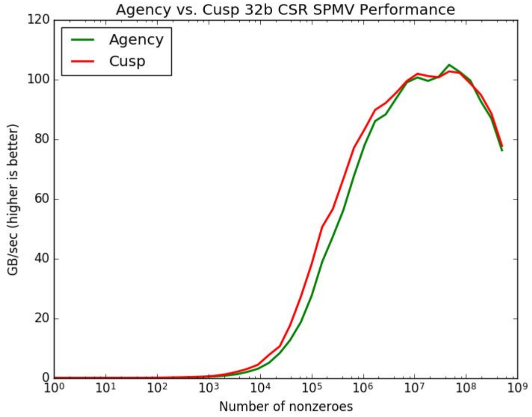
Transparent multi-GPU

```
// create an executor collecting all the GPUs in the system
cuda::multidevice_executor exec(cuda::all_devices());

// create some arrays spanning all the GPUs in the system
cuda::multidevice_array<float> x = ...
cuda::multidevice_array<float> y = ...
cuda::multidevice_array<float> z = ...

// execute only on GPU 0
saxpy(par.on(exec[0]), x, y, z);

// execute across all GPUs
saxpy(par.on(exec), x, y, z);
```

PERFORMANCE

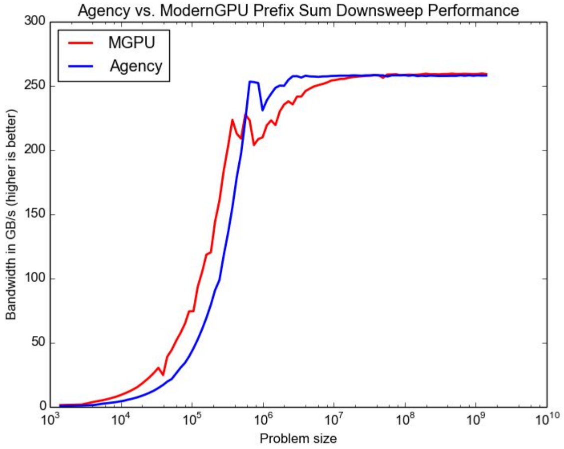
ABSTRACTIONS WITH MINIMAL OVERHEAD

Relative SpMV Bandwidth
on Test Matrices



SpMV performance
vs. hand-written CUDA
GeForce GTX 1070 (Pascal)

Prefix sum performance
vs. hand-written CUDA
Titan X (Kepler)



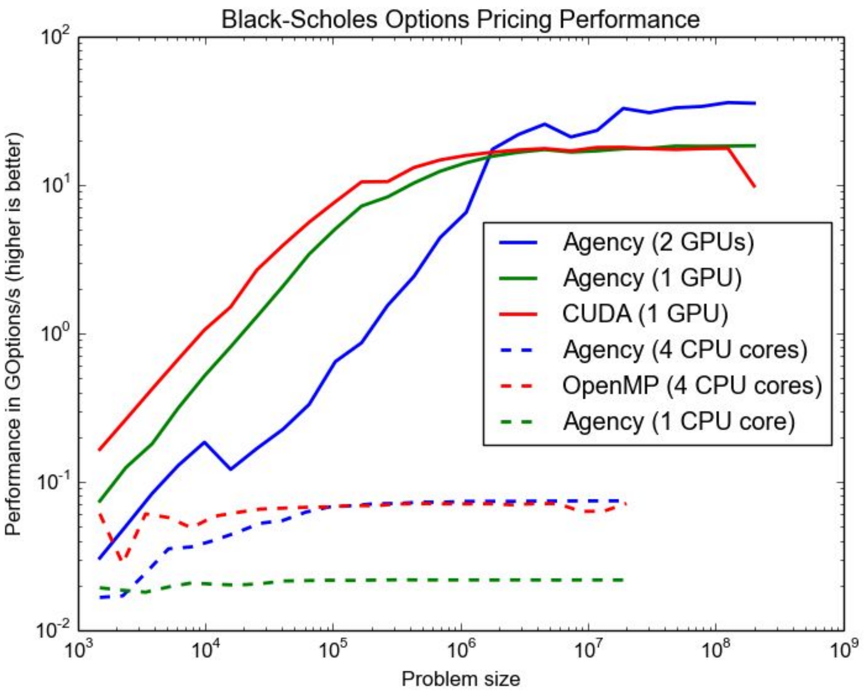
TARGET-INDEPENDENT CONSTRUCTS

Performance portability for *simple* memory access patterns

Write a portable parallel program

Choose execution target via executors

GPU execution with Unified Memory support provided by Pascal architecture



SINGLE-NODE HPGMG

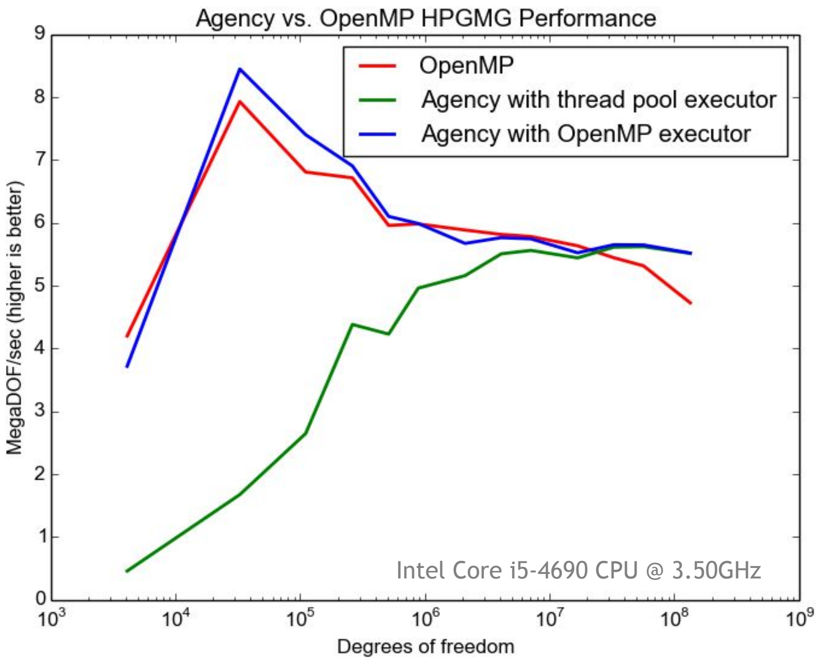
Agency CPU implementations vs. OpenMP-based original

Control behavior via choice of executor:

- using OpenMP runtime
- using pool of C++11 **`std::thread`**

Quite similar to OpenMP original when using OpenMP runtime

Reduction algorithm appears to account for better performance beyond 10^7 DoFs



SUMMARY

C++17 will deliver standard, high-level parallel algorithms

Agency provides efficient lower-level abstractions for parallel algorithms

Executors are low-level components for abstracting platform details

AVAILABLE ONLINE

Open source

On Github:

agency-library.github.io

Try with the latest CUDA:

developer.nvidia.com/cuda-toolkit

Quick start:

github.com/agency-library/agency/wiki/Quick-Start-Guide

NVIDIA