WRITING DATA PARALLEL ALGORITHMS ON GPUs

WITH C++ AMP

Ade Miller
Technical Director, CenturyLink Cloud.

ABSTRACT

- TODAY MOST PCS, TABLETS AND PHONES SUPPORT MULTI-CORE PROCESSORS AND MOST PROGRAMMERS HAVE SOME FAMILIARITY WITH WRITING (TASK) PARALLEL CODE. MANY OF THOSE SAME DEVICES ALSO HAVE GPUS BUT WRITING CODE TO RUN ON A GPU IS HARDER. OR IS IT?

  GETTING TO GRIPS WITH GPU PROGRAMMING IS REALLY ABOUT UNDERSTANDING THINGS IN A DATA PARALLEL WAY. THIS TALK WILL LOOK AT SOME OF THE COMMON PATTERNS FOR IMPLEMENTING ALGORITHMS ON TODAY'S GPUS USING EXAMPLES FROM THE C++ AMP ALGORITHMS LIBRARY. ALONG THE WAY IT WILL COVER SOME OF THE UNIQUE ASPECTS OF WRITING CODE FOR GPUS AND CONTRAST THEM WITH A MORE CONVENTIONAL CODE RUNNING ON A CPU.

I AM NOT A PROFESSIONAL

I DO THIS FOR FUN

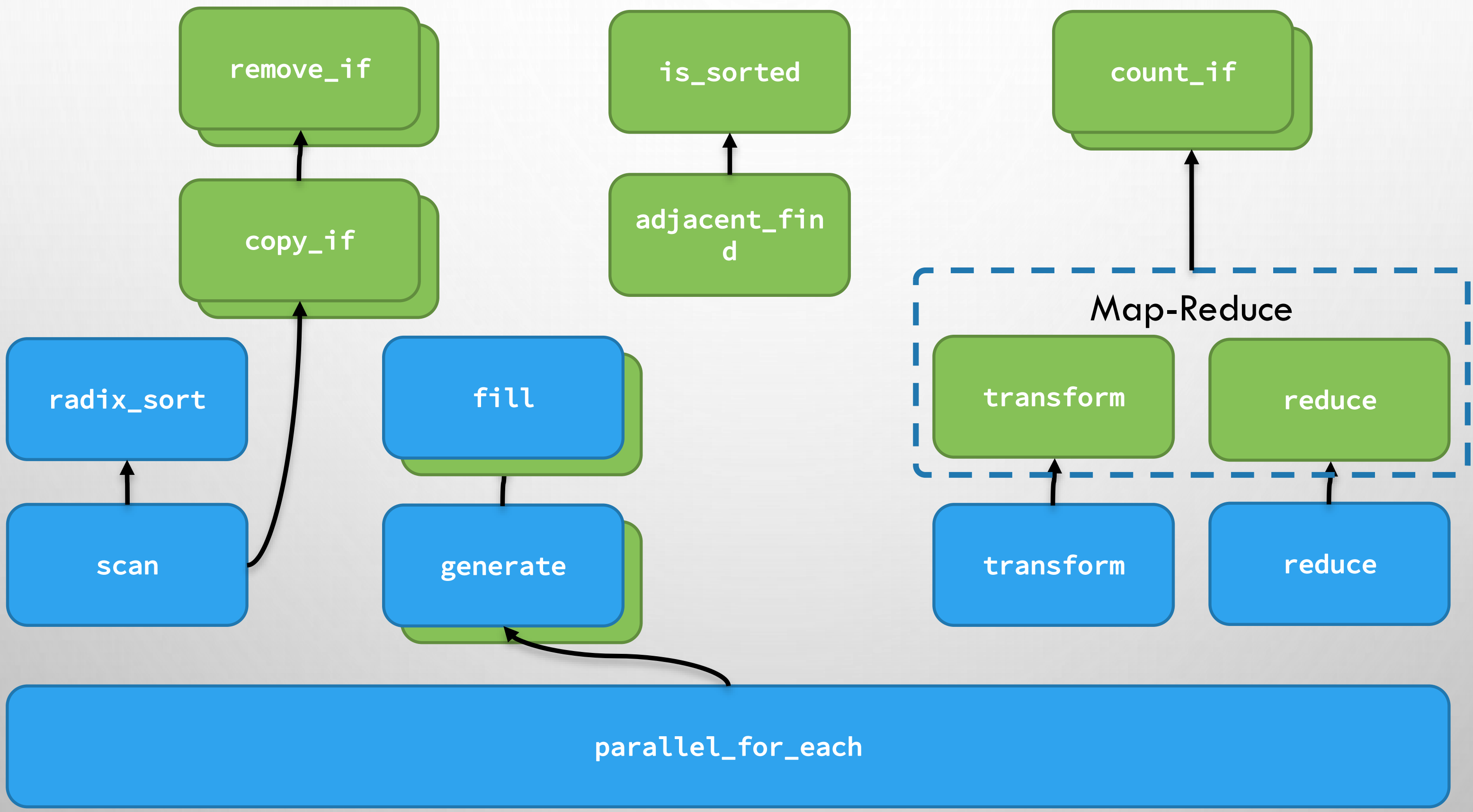
WHAT YOU'LL LEARN

- GIVE YOU A BETTER UNDERSTANDING OF HOW TO WRITE CODE FOR GPUs
- SOME MORE C++ AMP
- PASS ON SOME OF THE THINGS I LEARNT
  - WHEN WRITING THE BOOK, CASE STUDIES AND SAMPLES
  - DEVELOPING THE C++ AMP ALGORITHMS LIBRARY (AAL)

WHAT YOU'LL NOT LEARN

- HOW TO BE A PERFORMANCE GURU.
- ALL OF THE DEEP DARK SECRETS OF GPUs

ALGORITHM FAMILY TREE



TRANSFORM / MAP

GPU PROGRAMMING 101

TRANSFORM / MAP

$$i_n = f(i_n)$$

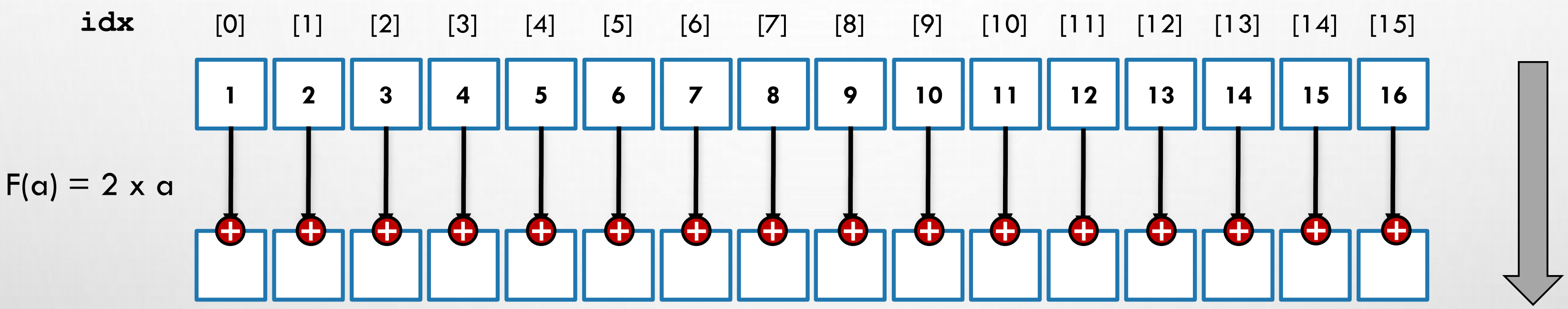
```
struct doubler_functor
{
    int operator()(const int& x) const { return x * 2; }
};

std::vector<int> input(1024);
std::iota(begin(input), end(input), 1);
std::vector<int> output(1024);

std::transform(begin(input), end(input), begin(output),
               doubler_functor());
```

TRANSFORM

1 → 1 MAPPINGS ARE TRIVIAL WITH GPUs



- Lines represent read or write memory accesses
- Red circles are an operation executed by a thread
- 22 Blue boxes are memory locations in <u>global</u> memory and their current value
- 25 Green boxes are memory locations in <u>tile</u> memory and their current value

TRANSFORM WITH AAL

```
struct doubler_functor
{
  int operator()(const int& x) const restrict(amp, cpu)
  {
    return x*2;
  }
};

std::vector<int> input(1024);
std::iota(begin(input), end(input), 1);
std::vector<int> output(1024);

concurrency::array_view<const int> input_av(input);
concurrency::array_view<int> output_av(output);
output_av.discard_data();

amp_stl_algorithms::transform(begin(input_av), end(input_av),
  begin(output_av), doubler_functor());
```

TRANSFORM UNDER THE HOOD

```
concurrency::array_view<const int> input_av(input);
concurrency::array_view<int> output_av(output);
output_av.discard_data();

auto doubler_func = doubler_functor();
concurrency::parallel_for_each(output_av.extent,
  [=](concurrency::index<1> idx) restrict(amp)
{
    output_av[idx] = doubler_func(input_av[idx]);
});
```

REDUCE

REDUCE

$$s = \sum_{n=0}^{N} a_n$$

```
std::vector<int> data(1024);
int s = 0;
for (int i : data)
{
  s += i;
}

s = std::accumulate(cbegin(data), cend(data), 0);
```

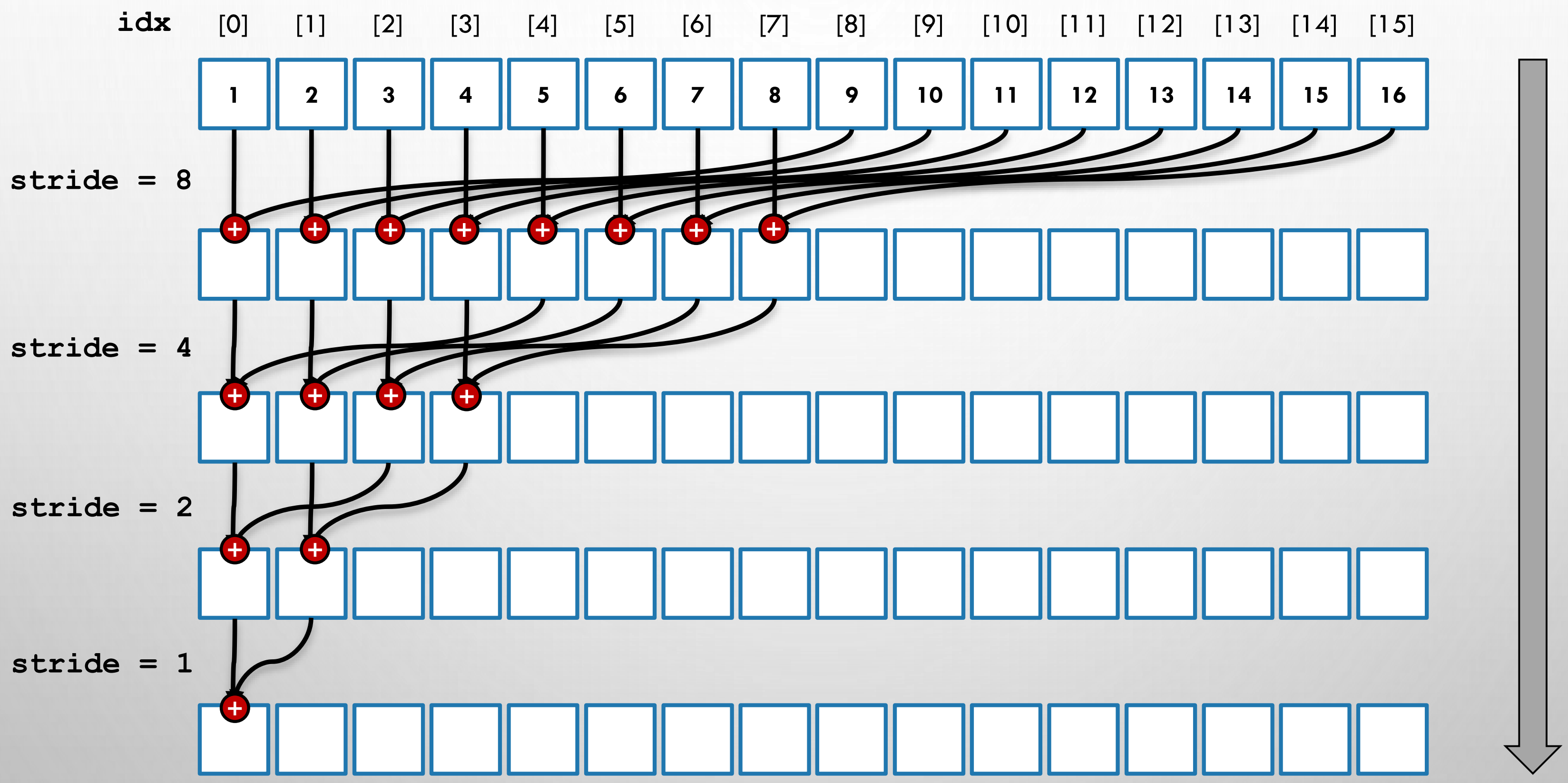
REMEMBER… $(\boldsymbol{a} + \boldsymbol{b}) + \boldsymbol{c} \neq \boldsymbol{a} + (\boldsymbol{b} + \boldsymbol{c})$

accumulate

≠

reduce

SIMPLE REDUCTION



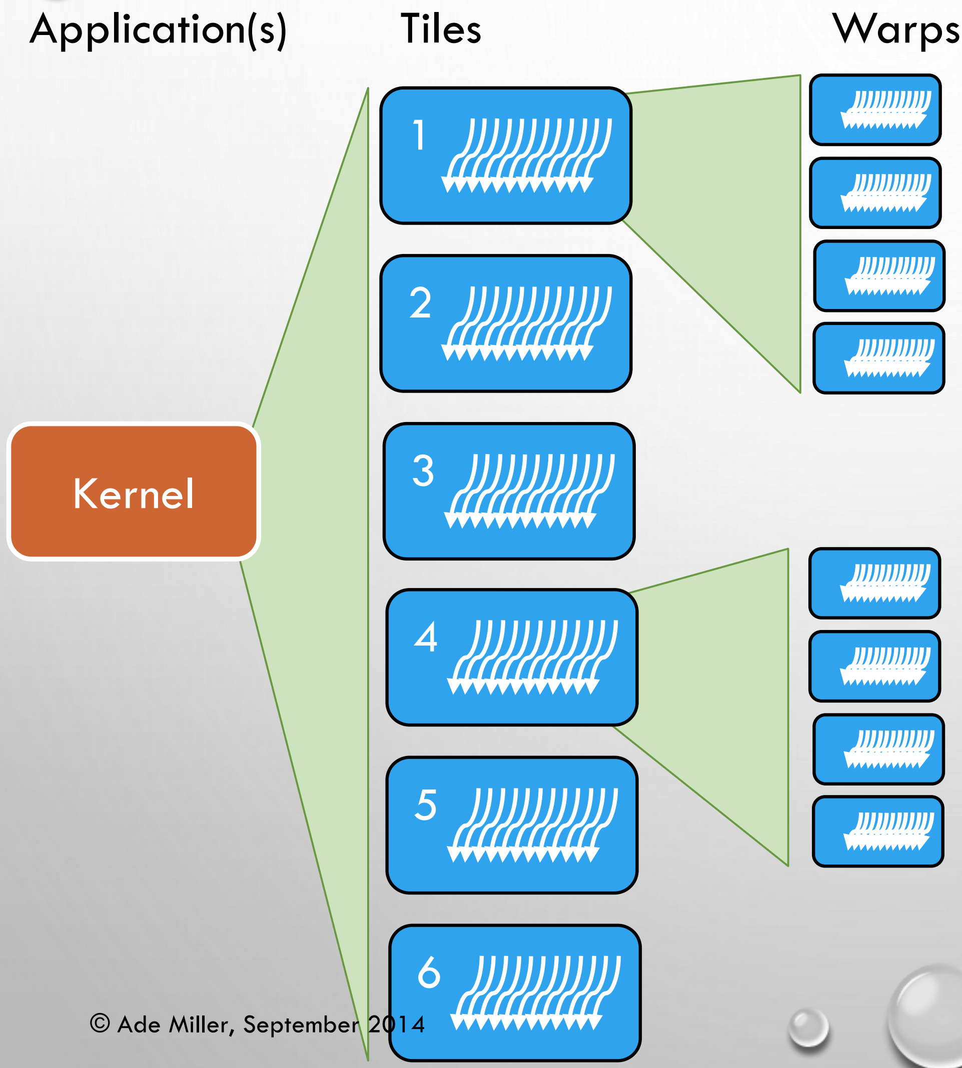
SIMPLE REDUCTION WITH C++ AMP

```
template <typename T>
int reduce_simple(const concurrency::array_view<T>& source) const
{
  int element_count = source.extent.size();
  std::vector<T> result(1);
  for (int stride = (element_count / 2); stride > 0; stride /= 2)
  {
    concurrency::parallel_for_each(concurrency::extent<1>(stride),
      [=](concurrency::index<1> idx) restrict(amp)
    {
      source[idx] += source[idx + stride];
    });
  }
  concurrency::copy(source.section(0, 1), result.begin());
  return result[0];
}
```

PROBLEMS WITH SIMPLE REDUCE

- DOESN'T USE TILE STATIC MEMORY
  - ALL READS AND WRITES ARE TO GLOBAL MEMORY
- MOST OF THE THREADS ARE IDLE MOST OF THE TIME

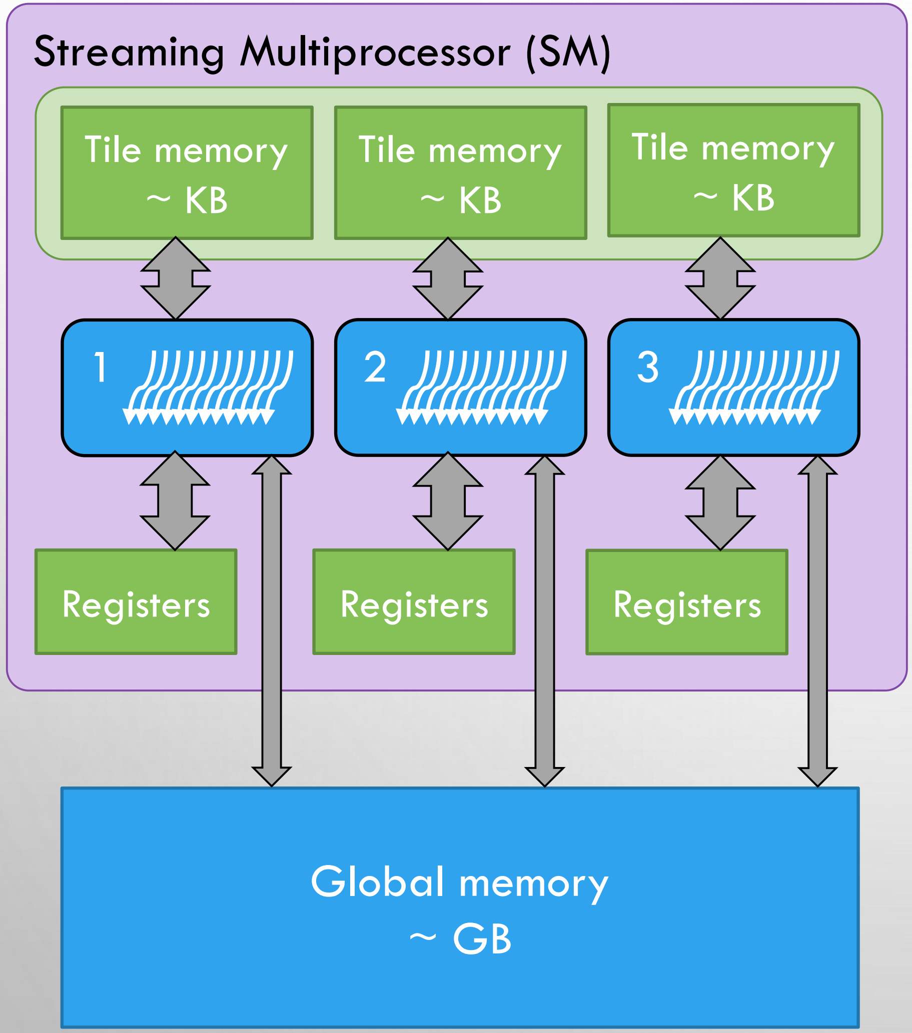
USING TILES AND TILE MEMORY: 1



- TILES ARE THE UNITS OF WORK USED TO SCHEDULE THE GPUs STREAMING MULTIPROCESSORS.
- THE SCHEDULER ALLOCATES WORK BASED ON THE AVAILABLE RESOURCES.
- EACH PROCESSOR FURTHER DIVIDES WORK UP INTO GROUPS OF THREADS CALLED WARPS.
- THE WARP SCHEDULER HIDES (MEMORY) LATENCY WITH COMPUTATION FROM OTHER CORES.

WARNING!
SIMPLIFIED VIEW

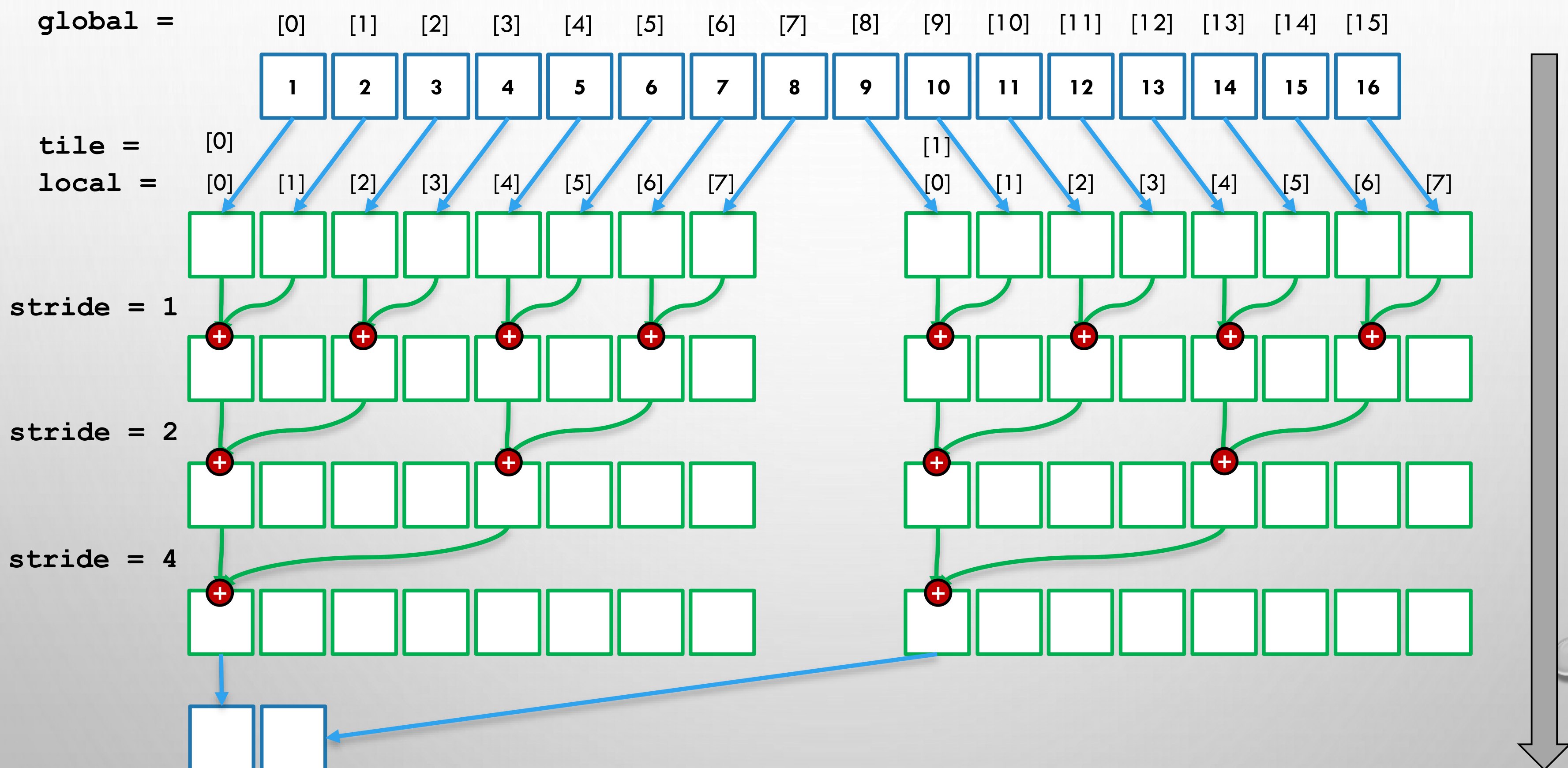
USING TILES AND TILE MEMORY: 2



- TILES CAN ACCESS GLOBAL MEMORY BUT WITH LIMITED SYNCHRONIZATION (ATOMIC OPERATIONS).
- TILES CAN ACCESS LOCAL TILE MEMORY USING SYNCHRONIZATION PRIMITIVES FOR COORDINATING TILE MEMORY ACCESS WITHIN A TILE.
- EACH SM ALSO HAS REGISTERS WHICH ARE USED BY THE TILES.
- YOUR APPLICATION MUST BALANCE RESOURCE USE TO MAXIMIZE (THREAD) OCCUPANCY.

WARNING! SIMPLIFIED VIEW

USE TILE MEMORY



A TYPICAL TILED KERNEL

```
template <typename T, int tile_size = 64>
int reduce_tiled(concurrency::array_view<T>& source) const
{
    int element_count = source.extent.size();
    concurrency::array<int, 1> per_tile_results(element_count / tile_size);
    concurrency::array_view<int, 1> per_tile_results_av(per_tile_results);
    per_tile_results_av.discard_data();

    while (element_count >= tile_size)
    {
        concurrency::extent<1> ext(element_count);
        concurrency::parallel_for_each(ext.tile<tile_size>(),
            [=](tiled_index<tile_size> tidx) restrict(amp)
        {
            int tid = tidx.local[0];
            tile_static int tile_data[tile_size];
            tile_data[tid] = source[tidx.global[0]];
            tidx.barrier.wait();

            for (int stride = 1; stride < tile_size; stride *= 2)
            {
                if (tid % (2 * stride) == 0)
                    tile_data[tid] += tile_data[tid + stride];
                tidx.barrier.wait_with_tile_static_memory_fence();
            }

            if (tid == 0)
                per_tile_results_av[tidx.tile[0]] = tile_data[0];
        });

        element_count /= tile_size;
        std::swap(per_tile_results_av, source);
        per_tile_results_av.discard_data();
    }

    std::vector<int> partialResult(element_count);
    concurrency::copy(source.section(0, element_count), partialResult.begin());
    source.discard_data();
    return std::accumulate(partialResult.cbegin(), partialResult.cend(), 0);
}
```

Create a (global) array for storing per-tile results.

Load data into tile_static memory & wait.

Calculate result for each tile & wait.

Write tile results back into (global) array.

Read the final tile results out and accumulate on the CPU.

A TYPICAL TILED KERNEL

```
template <typename T, int tile_size = 64>
int reduce_tiled(concurrency::array_view<T>& source) const
{
    int element_count = source.extent.size();
    concurrency::array<int, 1> per_tile_results(element_count / tile_size);
    concurrency::array_view<int, 1> per_tile_results_av(per_tile_results);
    per_tile_results_av.discard_data();

    while (element_count >= tile_size)
    {
        concurrency::extent<1> ext(element_count);
        concurrency::parallel_for_each(ext.tile<tile_size>(),
            [=](tiled_index<tile_size> tidx) restrict(amp)
        {
            int tid = tidx.local[0];
            tile_static int tile_data[tile_size];
            tile_data[tid] = source[tidx.global[0]];
            tidx.barrier.wait();

            for (int stride = 1; stride < tile_size; stride *= 2)
            {
                if (tid % (2 * stride) == 0)
                    tile_data[tid] += tile_data[tid + stride];
                tidx.barrier.wait_with_tile_static_memory_fence();
            }

            if (tid == 0)
                per_tile_results_av[tidx.tile[0]] = tile_data[0];
        });

        element_count /= tile_size;
        std::swap(per_tile_results_av, source);
        per_tile_results_av.discard_data();
    }

    std::vector<int> partialResult(element_count);
    concurrency::copy(source.section(0, element_count), partialResult.begin());
    source.discard_data();
    return std::accumulate(partialResult.cbegin(), partialResult.cend(), 0);
}
```

Create a (global) array for storing per-tile results.

Load data into tile_static memory & wait.

Calculate result for each tile & wait.

Write tile results back into (global) array.

Read the final tile results out and accumulate on the CPU.

A TYPICAL TILED KERNEL

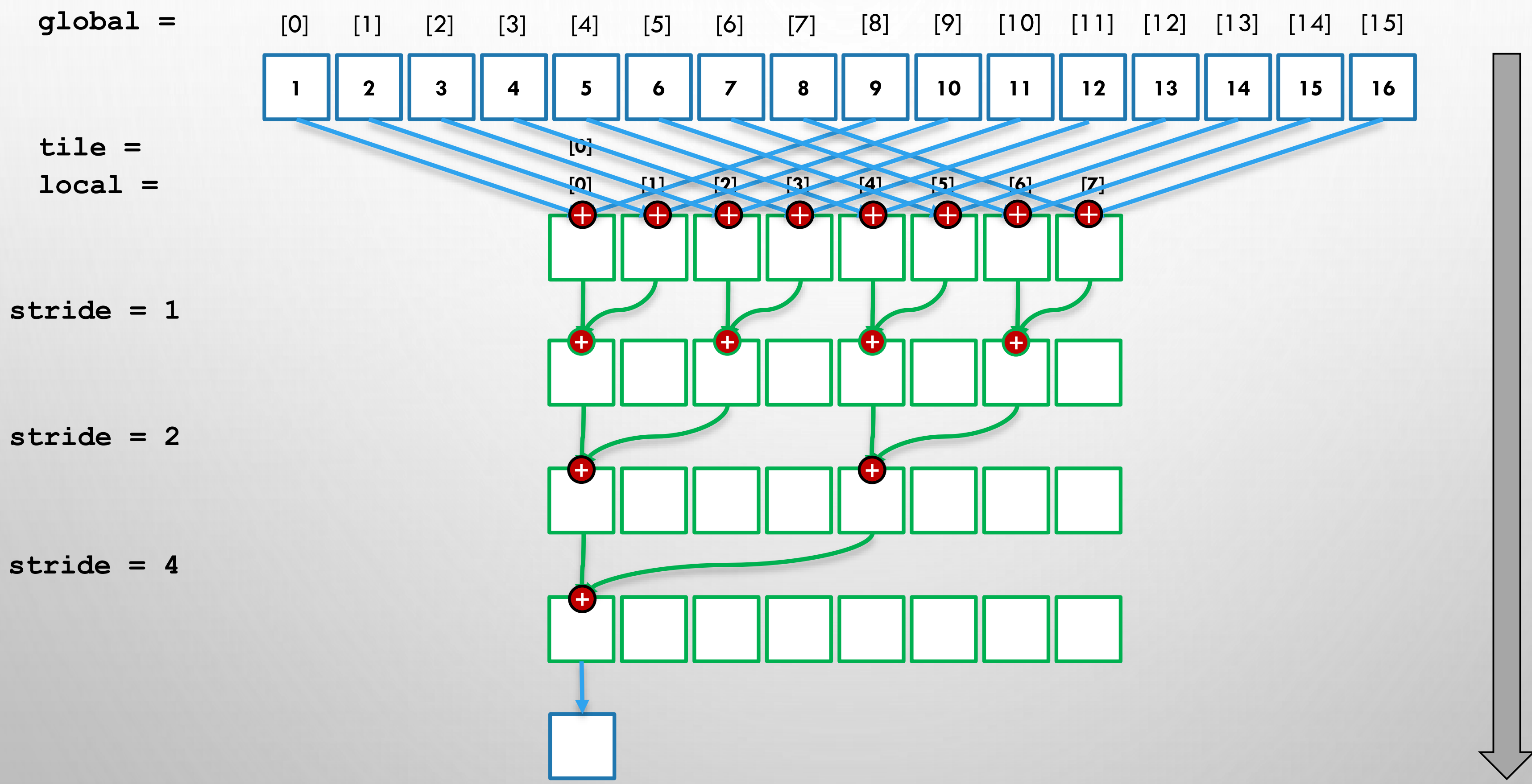
```
concurrency::extent<1> ext(element_count);
concurrency::parallel_for_each(ext.tile<tile_size>(),
[=](tiled_index<tile_size> tidx) restrict(amp)
{
  int tid = tidx.local[0];
  tile_static int tile_data[tile_size];
  tile_data[tid] = source[tidx.global[0]];
  tidx.barrier.wait();
  for (int stride = 1; stride < tile_size; stride *= 2)
  {
    if (tid % (2 * stride) == 0)
      tile_data[tid] += tile_data[tid + stride];
    tidx.barrier.wait_with_tile_static_memory_fence();
  }
  if (tid == 0)
    per_tile_results_av[tidx.tile[0]] = tile_data[0];
});
```

Each thread copies to tile_static memory and waits.

Each thread sums neighbors in tile_static memory and waits.

1st tile thread copies result to global memory.

REDUCE IDLE THREADS



REDUCE IDLE THREADS

```
concurrency::extent<1> ext(element_count / 2);

concurrency::parallel_for_each(ext.tile<tile_size>(),
    [=](tiled_index<tile_size> tidx) restrict(amp)
{
    int tid = tidx.local[0];
    tile_static int tile_data[tile_size];

    int rel_idx = tidx.tile[0] * (tile_size * 2) + tid;
    tile_data[tid] = source[rel_idx] + source[rel_idx + tile_size];

    tidx.barrier.wait();


    // Loop that does all the actual work...
});
```

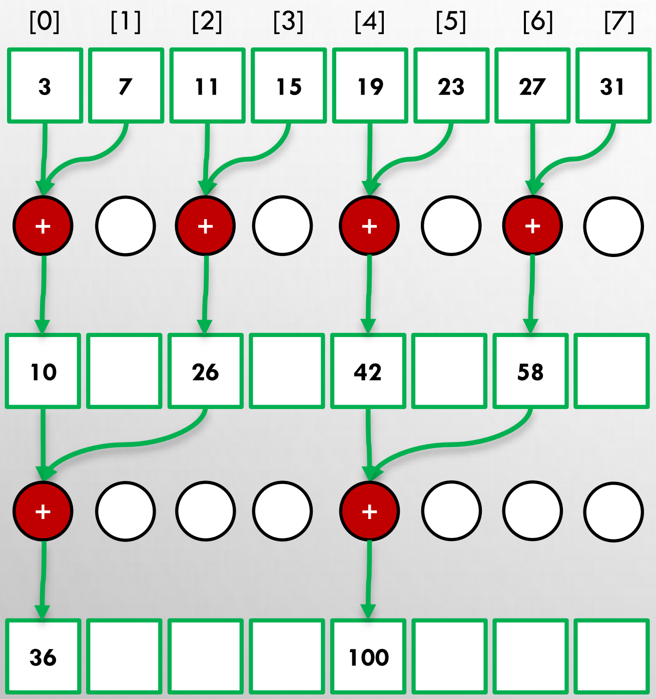
NEW PROBLEMS…

NOW THE KERNEL CONTAINS DIVERGENT CODE

- A TIGHT LOOP CONTAINING A CONDITIONAL

OUR MEMORY ACCESS PATTERNS ARE ALSO BAD

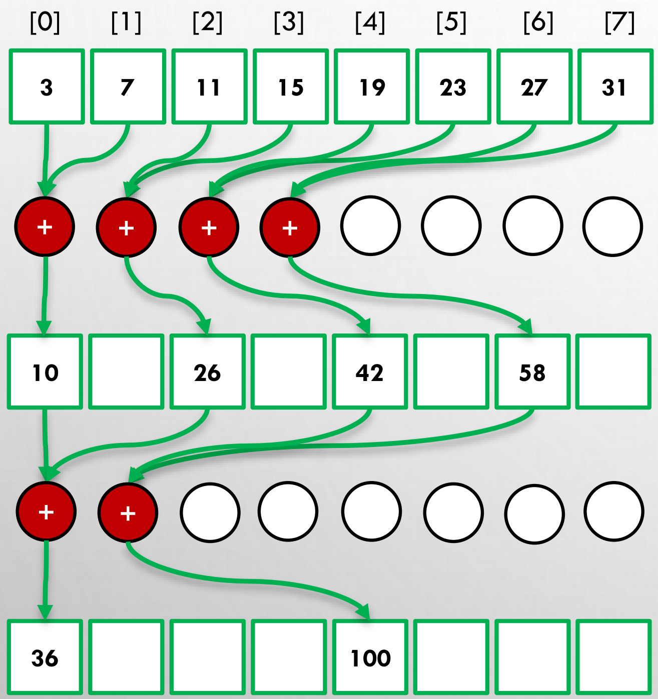
MINIMIZE DIVERGENT CODE




```
for(int stride =1; stride <tile_size; stride *=2)
{
  if (tid % (2 * stride) == 0)
    tile_data[tid] += tile_data[tid + stride];

  tidx.barrier
    .wait_with_tile_static_memory_fence();
}
```

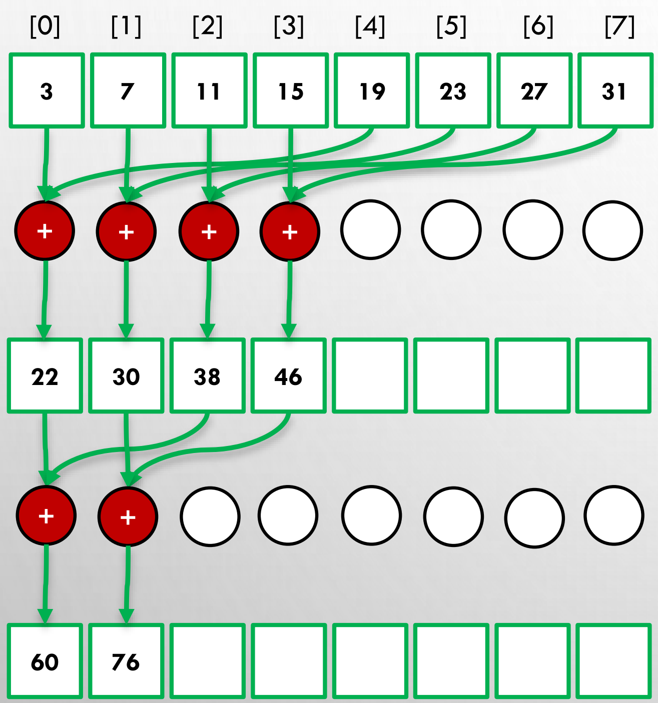
MINIMIZE DIVERGENT CODE




```
for(int stride =1; stride <tile_size; stride *=2)
{
  int index = 2 * stride * tid;
  if (index < tile_size)
    tile_data[index] += tile_data[index +stride];

  tidx.barrier
    .wait_with_tile_static_memory_fence();
}
```

REDUCE BANK CONFLICTS




```
for(int stride=(tile_size/2);stride>0;stride/=2)
{
  if (tid < stride)
    tileData[tid] += tileData[tid + stride];

  tidx.barrier
    .wait_with_tile_static_memory_fence();
}
```

REDUCTION THE EASY WAY…

THE AAL INCLUDES AN IMPLEMENTATION OF REDUCE.

```
concurrency::array_view<int> input_av(input_av);

int result =
  amp_algorithms::reduce(input_av, amp_algorithms::plus<int>());
```

THIS IS ANOTHER EXAMPLE OF WHY YOU SHOULD USE LIBRARIES WHERE POSSIBLE.

LET SOMEONE ELSE DO THE WORK FOR YOU!

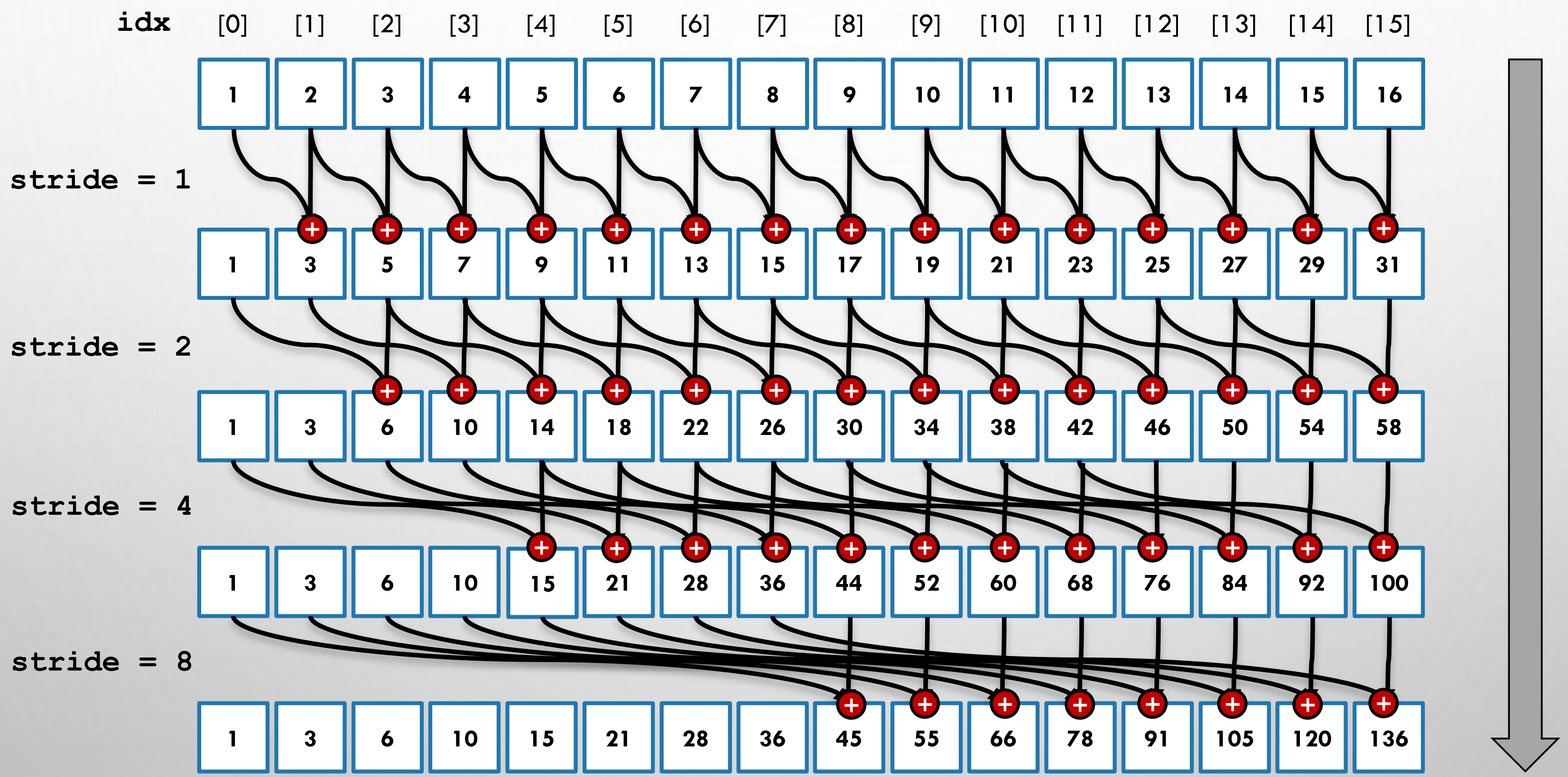
SCAN

INCLUSIVE SCAN

$$a_i = \sum_{n=0}^{i-1} a_n$$

```
std::vector<int> data(1024);
for (size_t i = 1; i < data.size(); ++i)
{
  data[i] += data[i - 1];
}
```

SIMPLE INCLUSIVE SCAN



PROBLEMS WITH SIMPLE INCLUSIVE SCAN

SEQUENTIAL SCAN IS **O**(N)

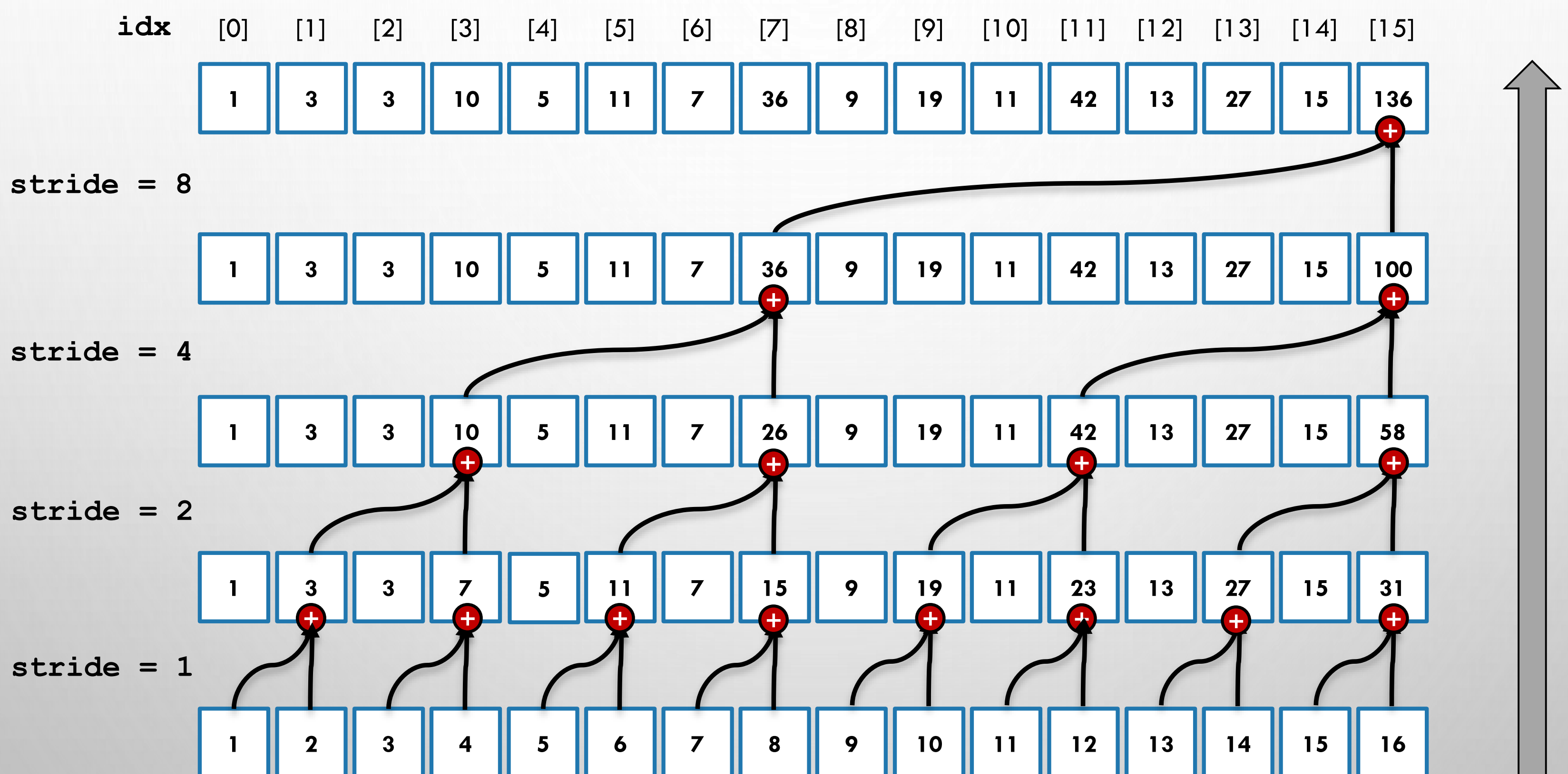
SIMPLE SCAN IS **O**(N LOG_2 N)

EXCLUSIVE SCAN

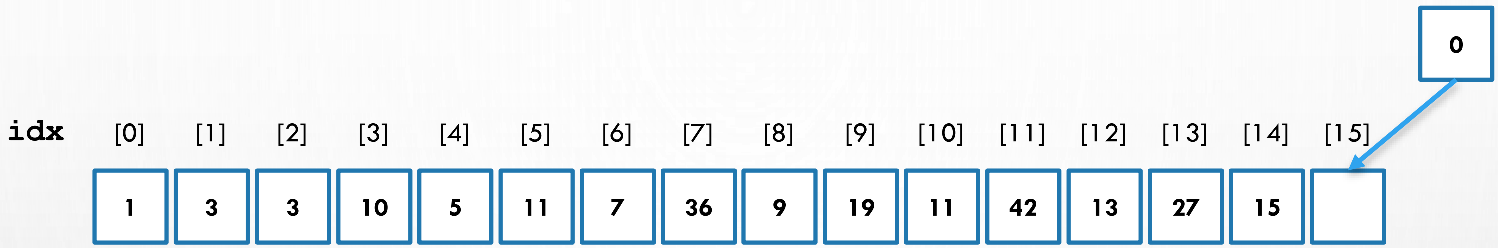
$$a_i = \sum_{n=0}^{i-1} a_n$$

```
std::vector<int> input(1024);
std::vector<int> output(1024);
output[0] = 0;
for (size_t i = 1; i < output.size(); ++i)
{
  output[i] = output[i - 1] + input[i - 1];
}
```

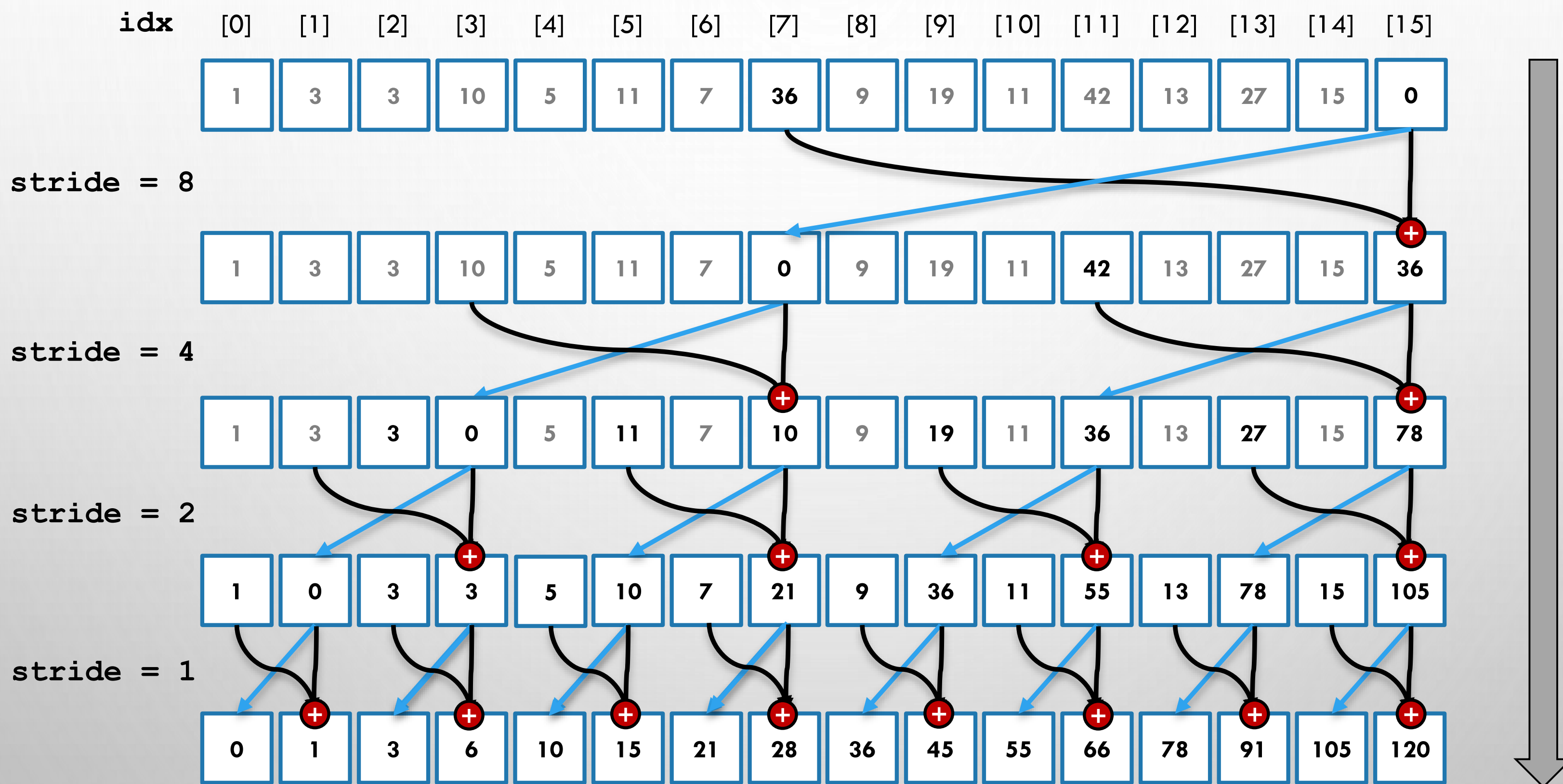
EXCLUSIVE SCAN STEP 1: UP-SWEEP



EXCLUSIVE SCAN STEP 2: DOWN-SWEEP



EXCLUSIVE SCAN STEP 2: DOWN-SWEEP



SCAN THE EASY WAY…

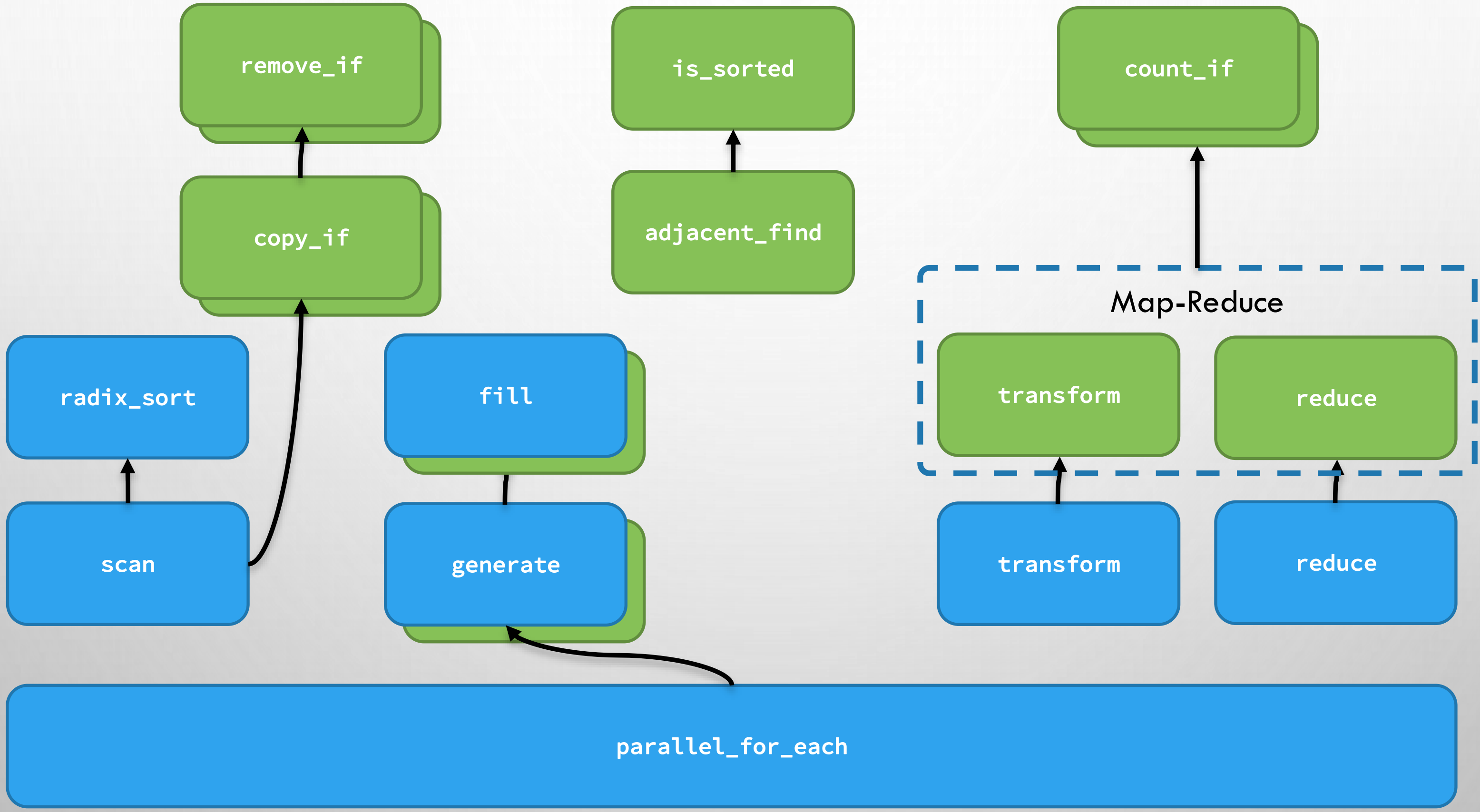
THE AAL INCLUDES A C++ AMP IMPLEMENTATION OF SCAN.

```
concurrency::array_view<int> input_av(input_vw);

scan<warp_size, scan_mode::exclusive>(input_vw, input_vw,
    amp_algorithms::plus<int>());
```

THERE IS ALSO A DIRECTX SCAN WRAPPER

ALGORITHM FAMILY TREE



SUMMARY

- MEMORY
  - MINIMIZE UNNECESSARY COPYING TO AND FROM THE GPU
  - MAKE USE OF TILE STATIC MEMORY WHERE POSSIBLE
  - THINK ABOUT MEMORY ACCESS PATTERNS IN BOTH GLOBAL AND TILE MEMORY

- COMPUTE
  - MINIMIZE THE DIVERGENCE IN YOUR CODE
  - REDUCE THE NUMBER OF STALLED OR IDLE THREADS
  - THINK VERY CAREFULLY BEFORE RESORTING TO ATOMIC OPERATIONS

WHAT'S NEXT

AMP LIBRARY

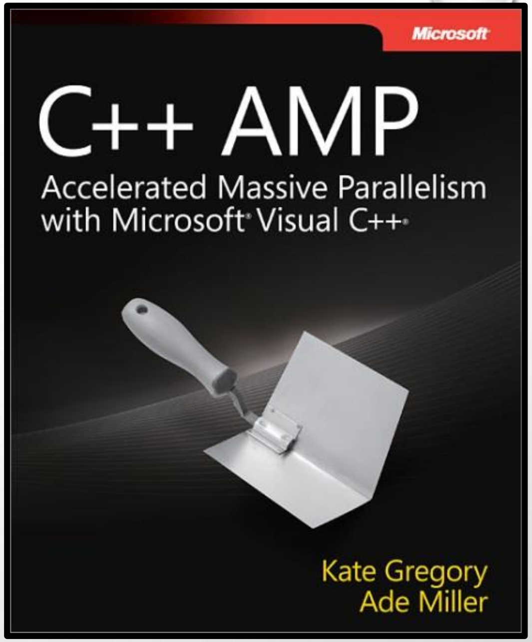
- **0.9.4:** RELEASE LATER THIS WEEK…
  - NUMEROUS NEW FEATURES INCLUDING; RADIX SORT & SCAN
- **0.9.5:** BY END OF YEAR (I HOPE)
  - PORTING TO RUN ON CLANG AND LLVM
  - ADDING SOME MORE FEATURES
  - SOME KEY PERFORMANCE TUNING

IF YOU WANT TO HELP I'M ALWAYS LOOKING FOR OSS DEVELOPERS

(THANKS TO AMD FOR PROVIDING SOME HARDWARE FOR THE LINUX PORT)

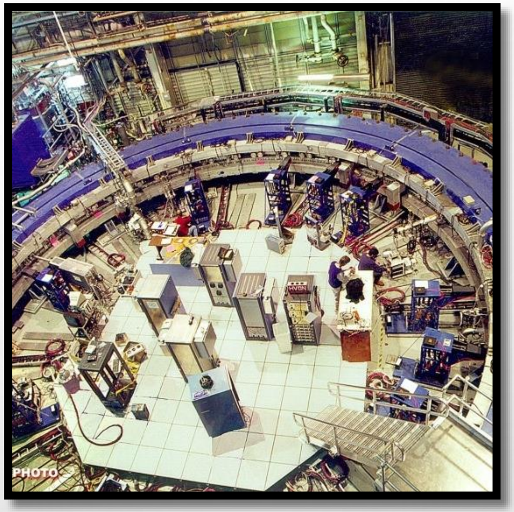
THE C++ AMP BOOK

- HTTP://WWW.GREGCONS.COM/CPPAMP
- WRITTEN BY KATE GREGORY & ADE MILLER
- COVERS ALL OF C++ AMP IN DETAIL, 350 PAGES
- ALL SOURCE CODE AVAILABLE ONLINE HTTP://AMPBOOK.CODEPLEX.COM/
- EBOOK ALSO AVAILABLE FROM O'REILLY BOOKS (OR AMAZON)



RESOURCES

- C++ AMP LIBRARY
  - HTTP://AMPALGORITHMS.CODEPLEX.COM/
- C++ AMP TEAM
  - BLOG: HTTP://BLOGS.MSDN.COM/B/NATIVECONCURRENCY/
  - SAMPLES: HTTP://BLOGS.MSDN.COM/B/NATIVECONCURRENCY/ARCHIVE/2012/01/30/C-AMP-SAMPLE-PROJECTS-FOR-DOWNLOAD.ASPX
- HSA FOUNDATION
  - HTTP://WWW.HSAFOUNDATION.COM/BRINGING-CAMP-BEYOND-WINDOWS-VIA-CLANG-LLVM/
- FORUMS TO ASK QUESTIONS
  - HTTP://STACKOVERFLOW.COM/QUESTIONS/TAGGED/C%2B%2B-AMP
  - HTTP://SOCIAL.MSDN.MICROSOFT.COM/FORUMS/EN/PARALLELCPPNATIVE/THREADS



QUESTIONS?