CS 677: Parallel Programming for Many-core Processors
Lecture 9

Instructor: Philippos Mordohai
Webpage: mordohai.github.io
E-mail: Philippos.Mordohai@stevens.edu

Outline

- Pinned Memory
- Streams
- Thrust

Pinned Memory

- *Page-locked* or *pinned* memory transfers attain the highest bandwidth between host and device
  - Ensures that host buffer does not get moved to virtual memory
- Allocated using the `cudaMallocHost()`
- Pinned memory should not be overused
  - Excessive use can reduce overall system performance
  - How much is too much is difficult to tell in advance

Asynchronous Transfers and Overlapping Transfers with Computation

- Data transfers between host and device using `cudaMemcpy()` are blocking transfers
  - Control is returned to the host thread only after the data transfer is complete.
- The `cudaMemcpyAsync()` function is a nonblocking variant of `cudaMemcpy()`
  - Unlike `cudaMemcpy()` the asynchronous transfer version requires pinned host memory

Asynchronous Transfers and Overlapping Transfers with Computation

```
cudaMemcpyAsync(a_d, a_h, size,
  cudaMemcpyHostToDevice, stream);
kernel<<<grid, block>>>(a_d);
cpuFunction();
```

- Memory transfer and device execution are performed in parallel with host execution
- Last argument of `cudaMemcpyAsync()` specifies stream
  - 0 is the default – only nonzero streams are asynchronous (more details soon)
  - Kernel does not begin execution until memory transfer is complete

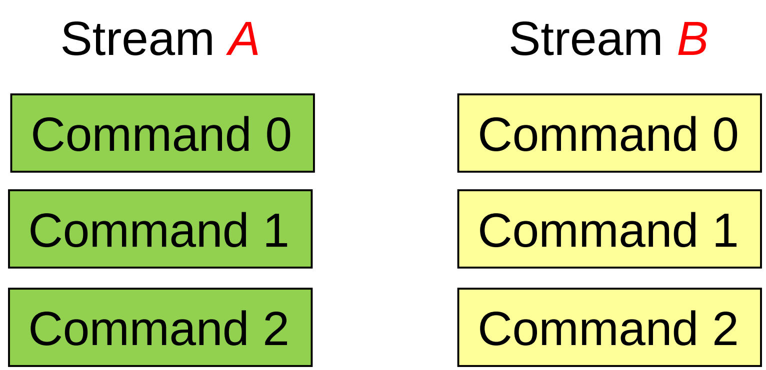
CUDA Streams

Patrick Cozzi
University of Pennsylvania
CIS 565 - Spring 2011

Steve Rennich
NVIDIA

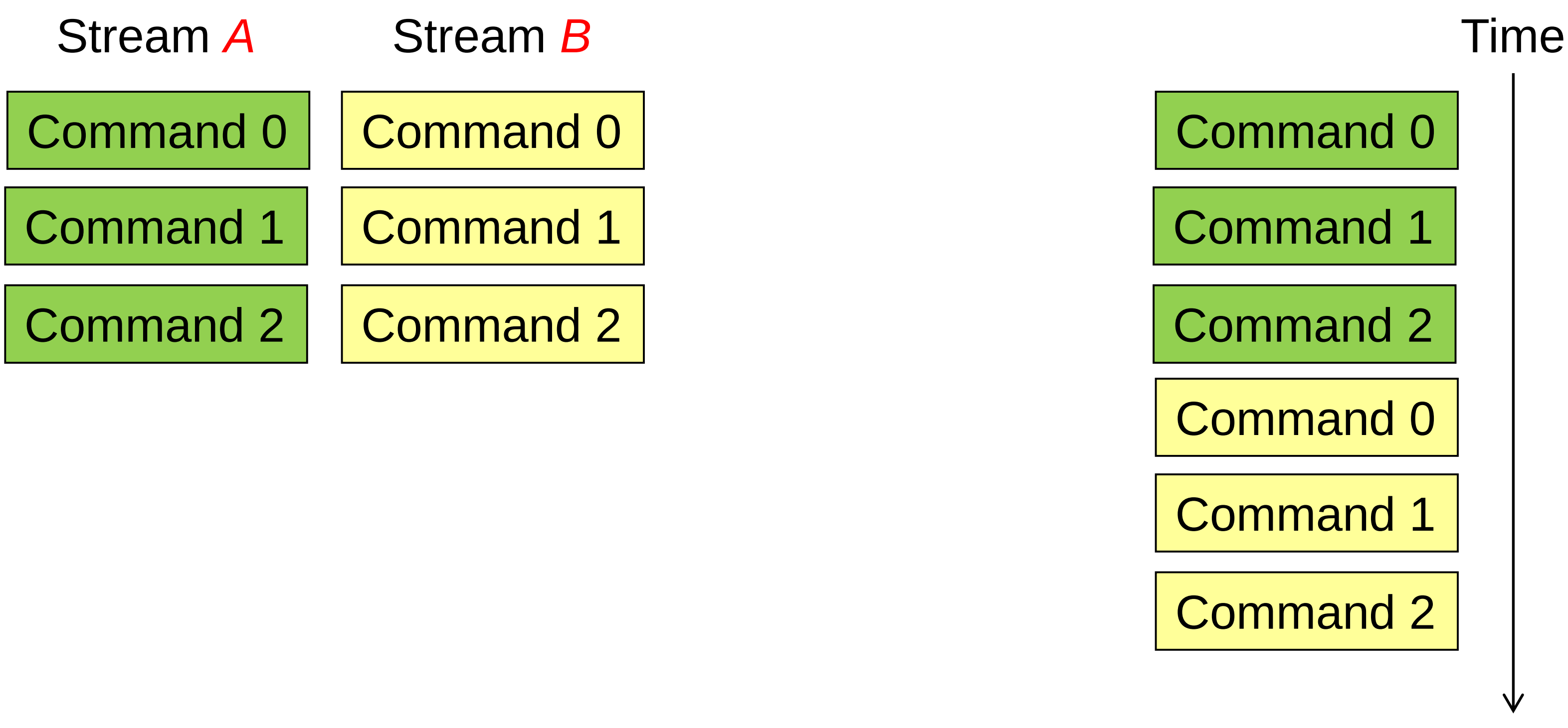
Streams

- **Stream**: Sequence of commands that execute in order
- Streams may execute their commands out-of-order or concurrently with respect to other streams

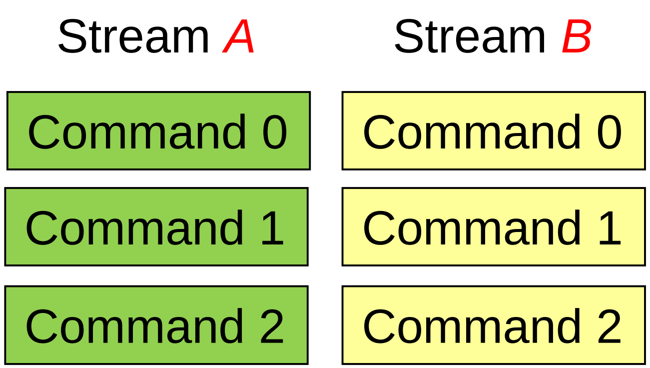


Streams

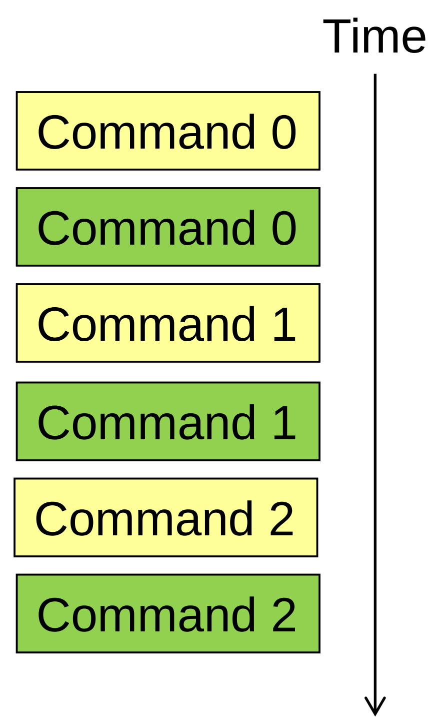
- Is this a possible order?



Streams

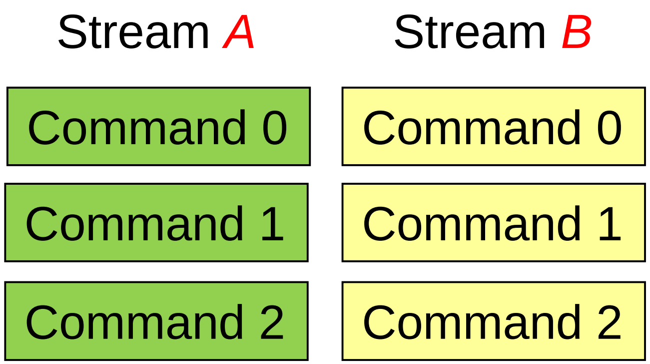


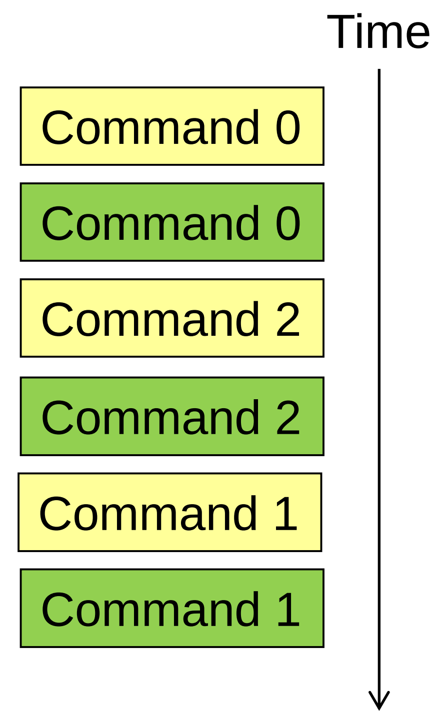
- Is this a possible order?



Streams

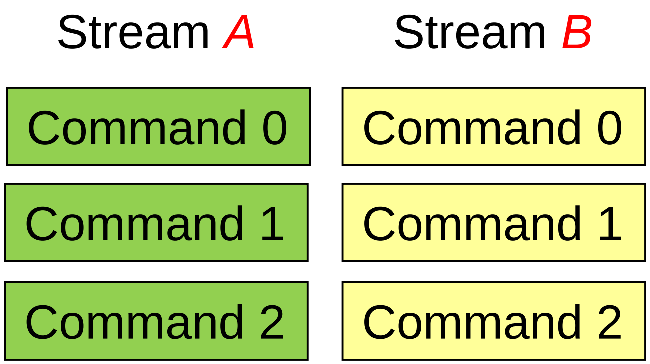
- Is this a possible order?

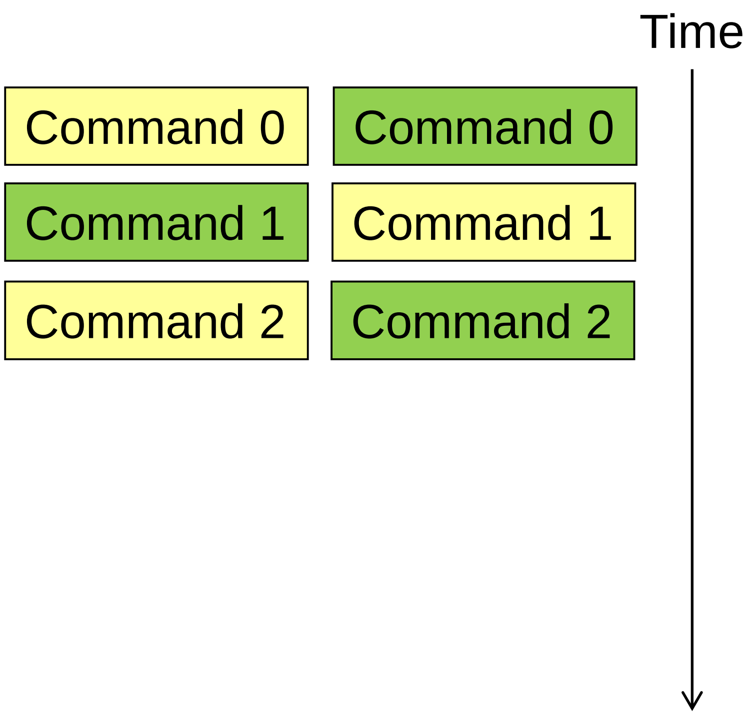




Streams

- Is this a possible order?

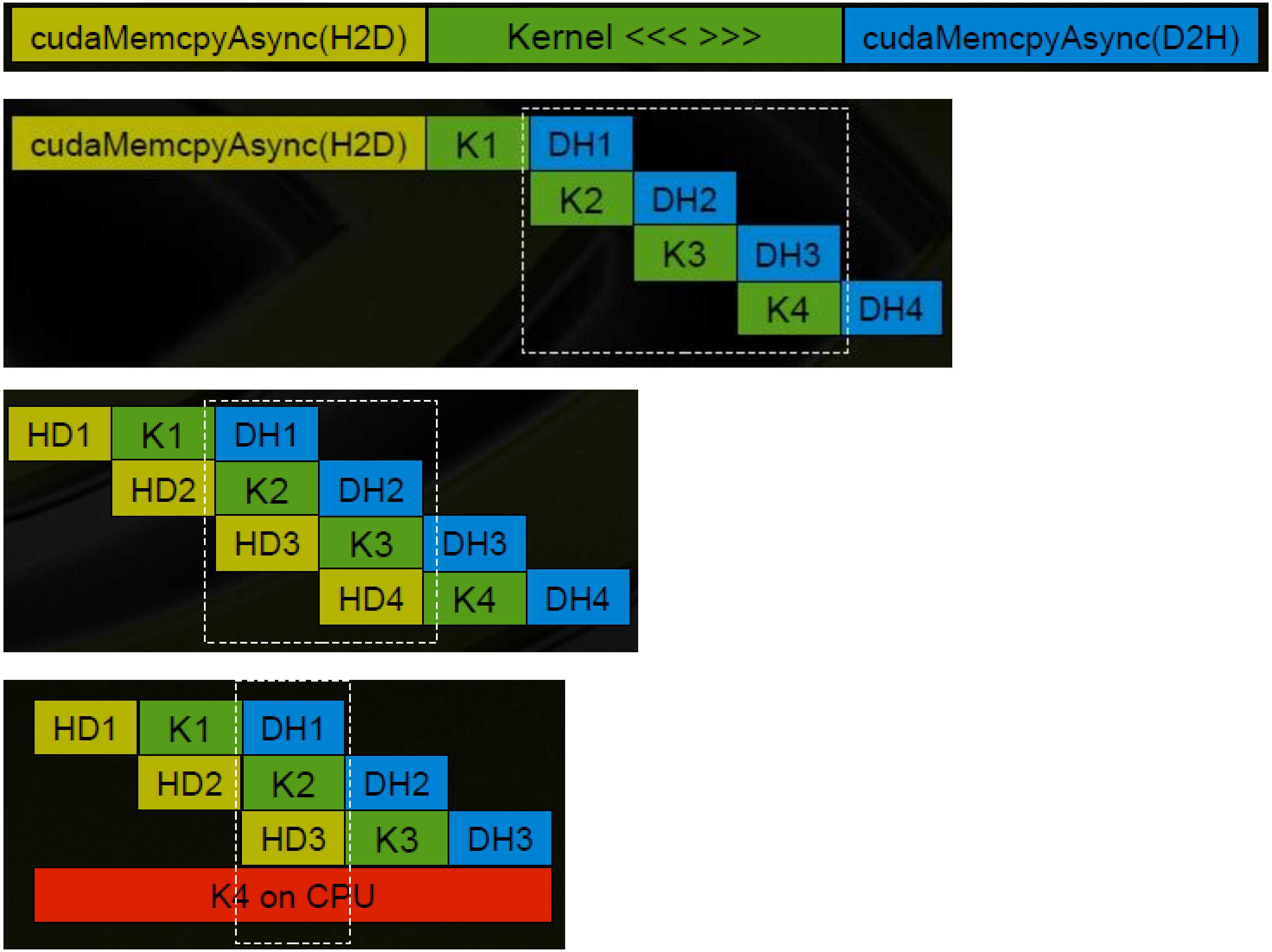




Streams

- In CUDA, what commands go in a stream?
  - Kernel launches
  - Host ⟺ device memory transfers

Amount of Concurrency



Default Stream (Stream ‘0’)

- Stream used when no stream is specified
- Completely synchronous w.r.t. host and device
  - As if cudaDeviceSynchronize() inserted before and after every CUDA operation
- Exceptions – asynchronous w.r.t. host
  - Kernel launches in the default stream
  - cudaMemcpy*Async
  - cudaMemset*Async
  - cudaMemcpy within the same device
  - H2D cudaMemcpy of 64kB or less

Requirements for Concurrency

- CUDA operations must be in different, non-0, streams
- cudaMemcpyAsync with host from 'pinned' memory
  - Page-locked memory
  - Allocated using cudaMallocHost() or cudaHostAlloc() (*)
- Sufficient resources must be available
  - cudaMemcpyAsyncs in different directions
  - Device resources (SMEM, registers, blocks, etc.)

* Both commands are roughly equivalent, but not in all versions of CUDA

Streams

- Code Example
  1. Create two streams
  2. Each stream:
     1. Copy page-locked memory to device
     2. Launch kernel
     3. Copy memory back to host
  3. Destroy streams

Stream Example (Step 1 of 3)

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
{
    cudaStreamCreate(&stream[i]);
}
```

Create two streams

```
float *hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Stream Example (Step 1 of 3)

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
{
    cudaStreamCreate(&stream[i]);
}

float *hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

Allocate two buffers in page-locked memory

Stream Example (Step 2 of 3)

```
for (int i = 0; i < 2; ++i)
{
  cudaMemcpyAsync(/* ... */,
    cudaMemcpyHostToDevice, stream[i]);
  kernel<<<100, 512, 0, stream[i]>>>
    (/* ... */);
  cudaMemcpyAsync(/* ... */,
    cudaMemcpyDeviceToHost, stream[i]);
}
```

Commands are assigned to, and executed by streams

Stream Example (Step 3 of 3)

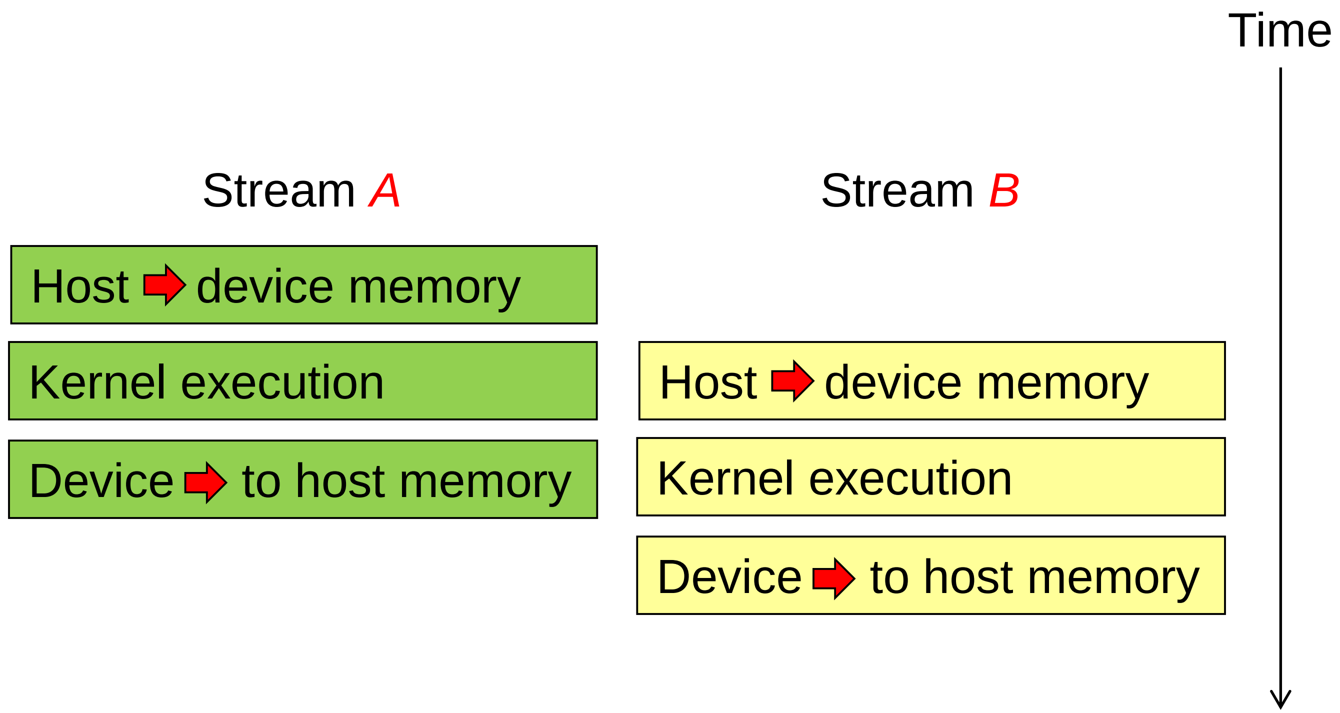
```
for (int i = 0; i < 2; ++i)
{
  // Blocks until commands complete
  cudaStreamDestroy(stream[i]);
}
```

Streams

- Assume compute capability 1.1 and above:
  - Overlap of data transfer and kernel execution
  - Concurrent kernel execution
  - Concurrent data transfer
- How can the streams overlap?

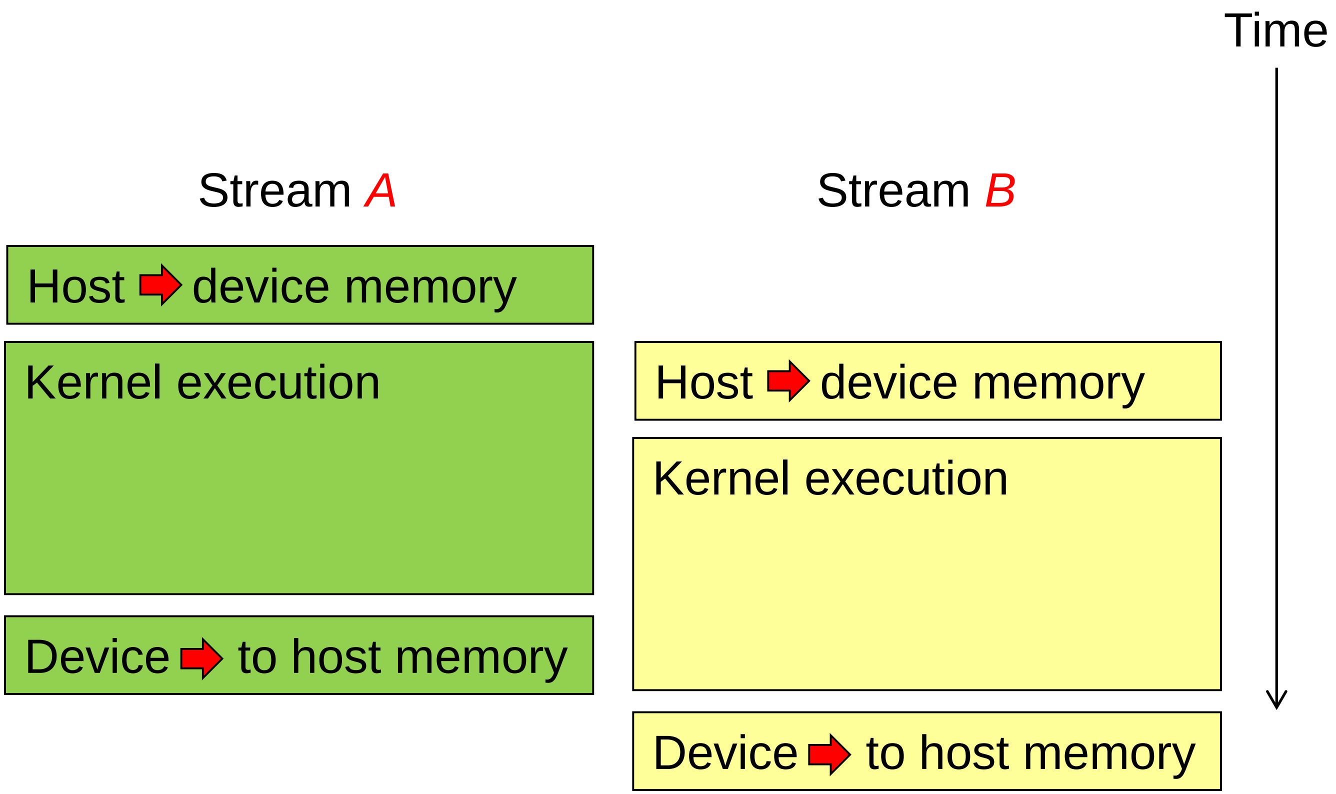
Streams

- Can we have more overlap than this?



Streams

- Can we have this?



Streams

- *Implicit Synchronization*
  - An operation that requires a dependency check to see if a kernel finished executing:
    - *Blocks* all kernel launches *from any stream* until the checked kernel is finished

- cudaStreamQuery() can be used to test if a stream has completed all operations

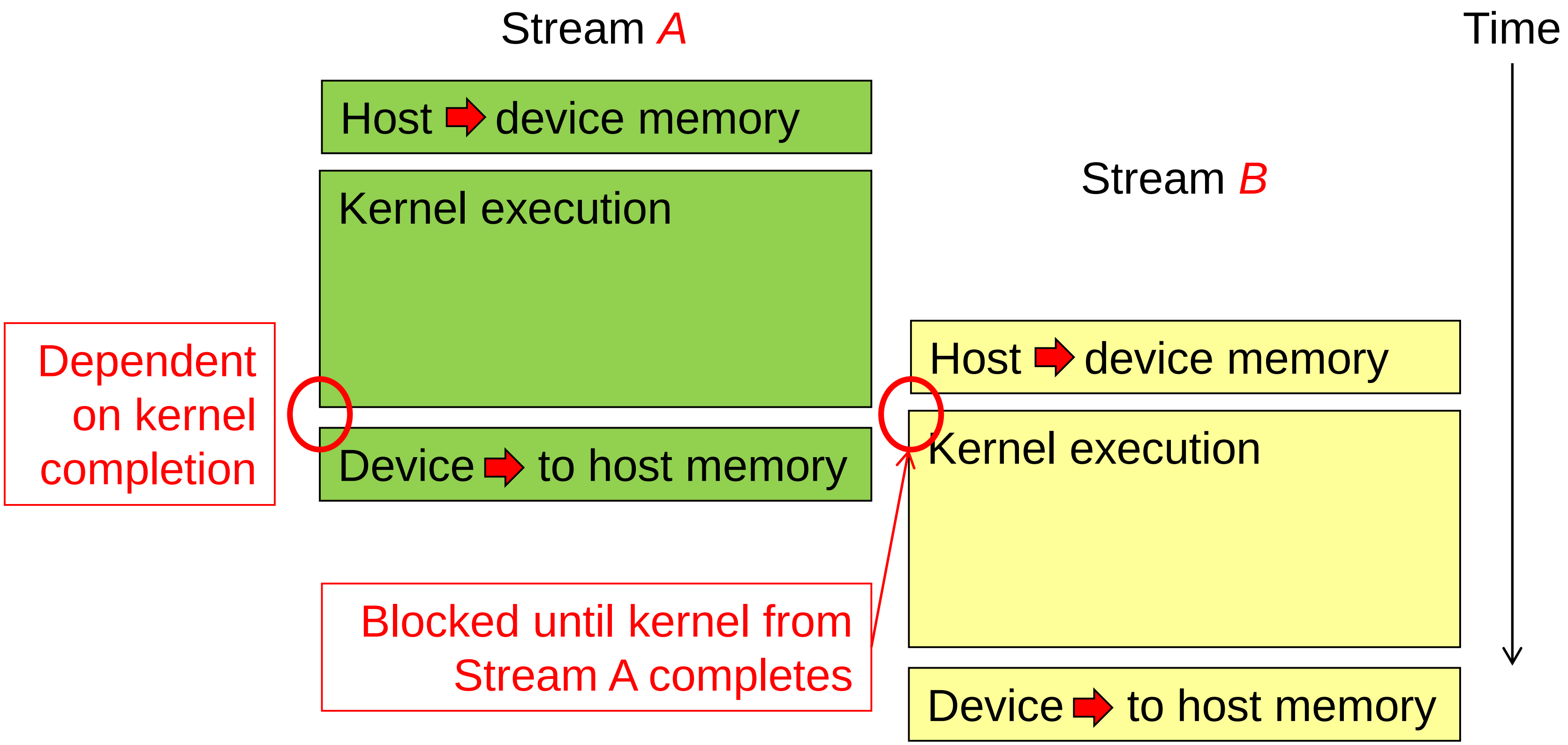
See 3.2.6.5.3 in the NVIDIA CUDA C Programming Guide for all limitations (version 3.2)

Implicit Synchronization

- These operations implicitly synchronize all other CUDA operations
  - Page-locked memory allocation
    - cudaMallocHost
    - cudaHostAlloc
  - Device memory allocation
    - cudaMalloc
  - Non-Async version of memory operations
    - cudaMemcpy* (no Async suffix)
    - cudaMemset* (no Async suffix)
  - Change to L1/shared memory configuration
    - cudaDeviceSetCacheConfig

Streams

- Can we have this?



Streams

- Performance Advice
  - Issue all independent commands before dependent ones
  - Delay synchronization (implicit or explicit) as long as possible

Streams

- Rewrite this to allow concurrent kernel execution

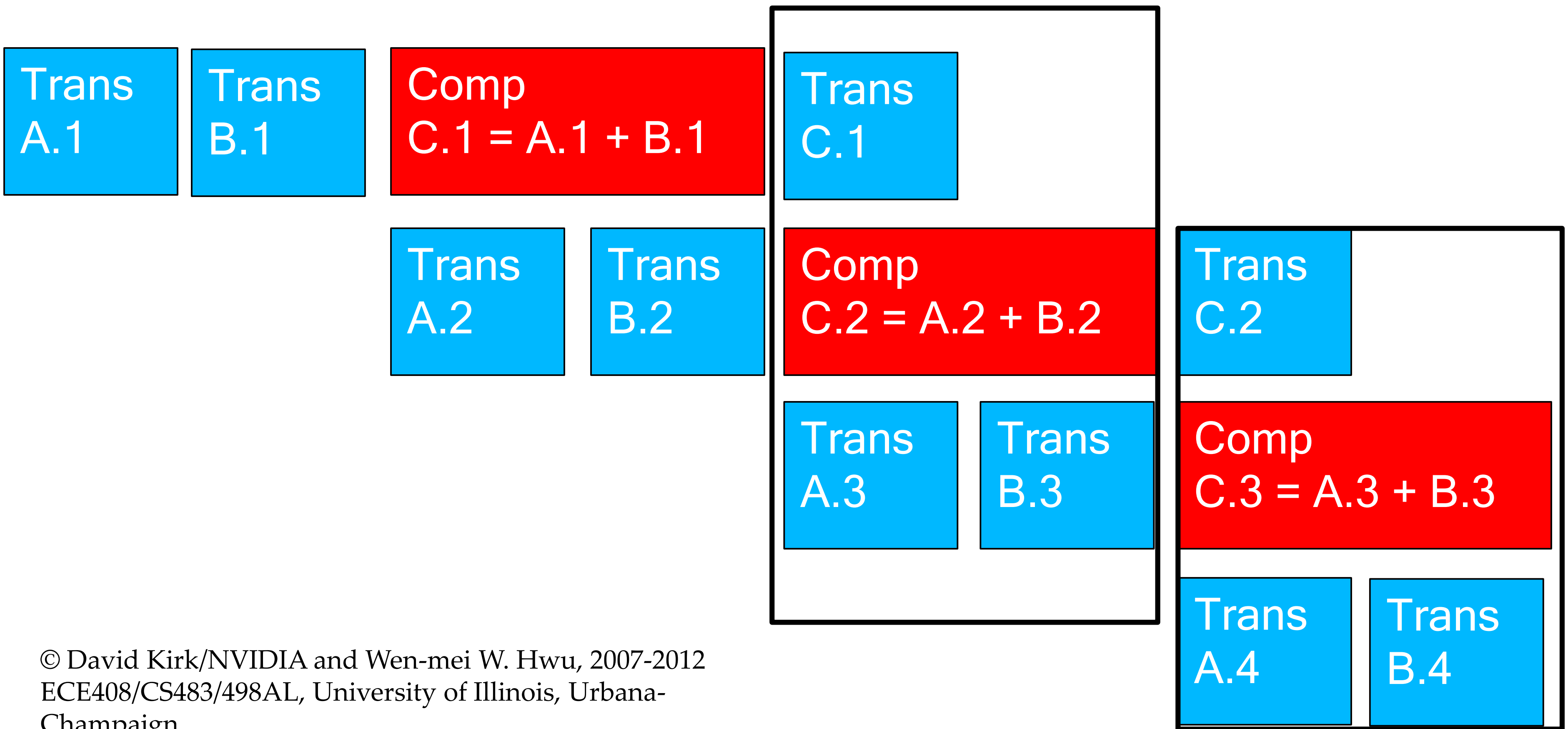
```
for (int i = 0; i < 2; ++i)
{
  cudaMemcpyAsync(/* ... */, stream[i]);
  kernel<<< /*... */ stream[i]>>>();
  cudaMemcpyAsync(/* ... */, stream[i]);
}
```

Streams

```
for (int i = 0; i < 2; ++i) // to device
 cudaMemcpyAsync(/* ... */, stream[i]);

for (int i = 0; i < 2; ++i)
  kernel<<< /*... */ stream[i]>>>();

for (int i = 0; i < 2; ++i)  // to host
  cudaMemcpyAsync(/* ... */, stream[i]);
```

Overlapped (Pipelined) Timing

- Divide large vectors into segments
- Overlap transfer and compute of adjacent segments



Explicit Syncrhonization

- `cudaDeviceSynchronize()`
  - Blocks until commands in all streams finish
  - cudaThreadSynchrhonize() has been deprecated
- `cudaStreamSynchronize(streamid)`
  - Blocks until commands in a specific stream finish

Synchronization using Events

- Create specific 'Events', within streams, to use for synchronization
- `cudaEventRecord ( event, streamid )`
- `cudaEventSynchronize ( event )`
- `cudaStreamWaitEvent ( stream, event )`
- `cudaEventQuery ( event )`

Explicit Synchronization Example

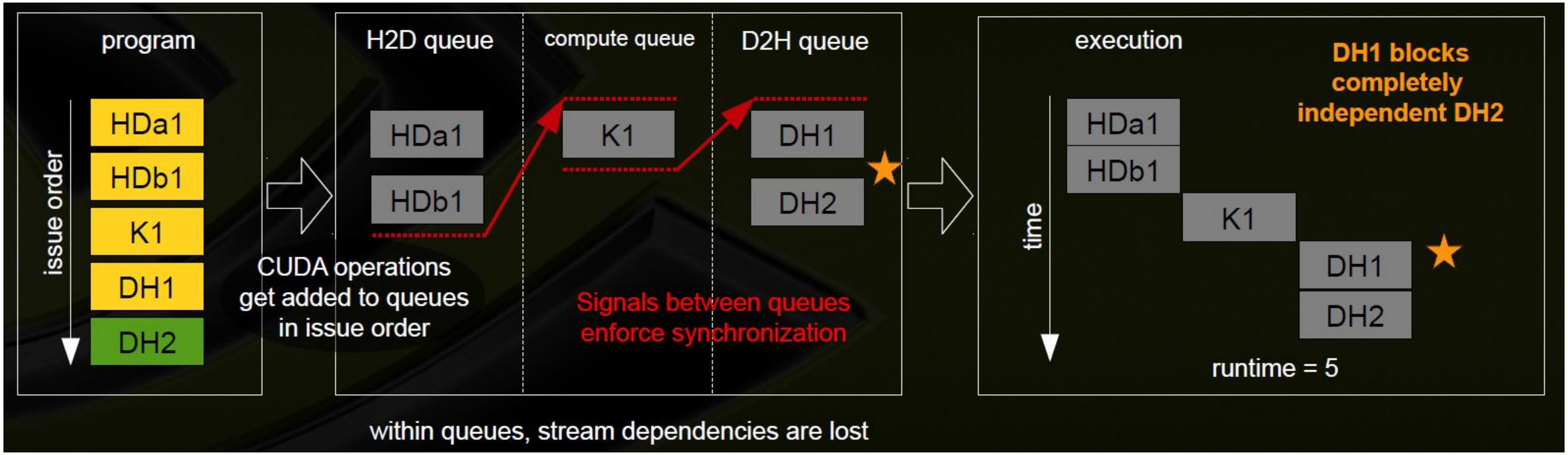
```
{
    cudaEvent_t event;
    cudaEventCreate (&event); // create event

     // 1) H2D copy of new input
    cudaMemcpyAsync ( d_in, in, size, H2D, stream1 );
    cudaEventRecord (event, stream1);                    // record event

    // 2) D2H copy of previous result
    cudaMemcpyAsync ( out, d_out, size, D2H, stream2 );
    cudaStreamWaitEvent ( stream2, event );              // wait for event in stream1

    kernel <<< , , , stream2 >>> ( d_in, d_out );        // 3) must wait for 1 and 2
    asynchronousCPUmethod ( … );
}
```

Stream Scheduling

- Fermi hardware has 3 queues
  - 1 Compute Engine queue
  - 2 Copy Engine queues - one for H2D and one for D2H
- CUDA operations are dispatched to HW in the sequence they were issued
  - Placed in the relevant queue
  - Stream dependencies between engine queues are maintained, but lost within an engine queue
- A CUDA operation is dispatched from the engine queue if:
  - Preceding calls in the same stream have completed,
  - Preceding calls in the same queue have been dispatched, and
  - Resources are available
- CUDA kernels may be executed concurrently if they are in different streams
  - Threadblocks for a given kernel are scheduled if all threadblocks for preceding kernels have been scheduled and there still are SM resources available
- Note that a blocked operation blocks all other operations in the queue, even in other streams

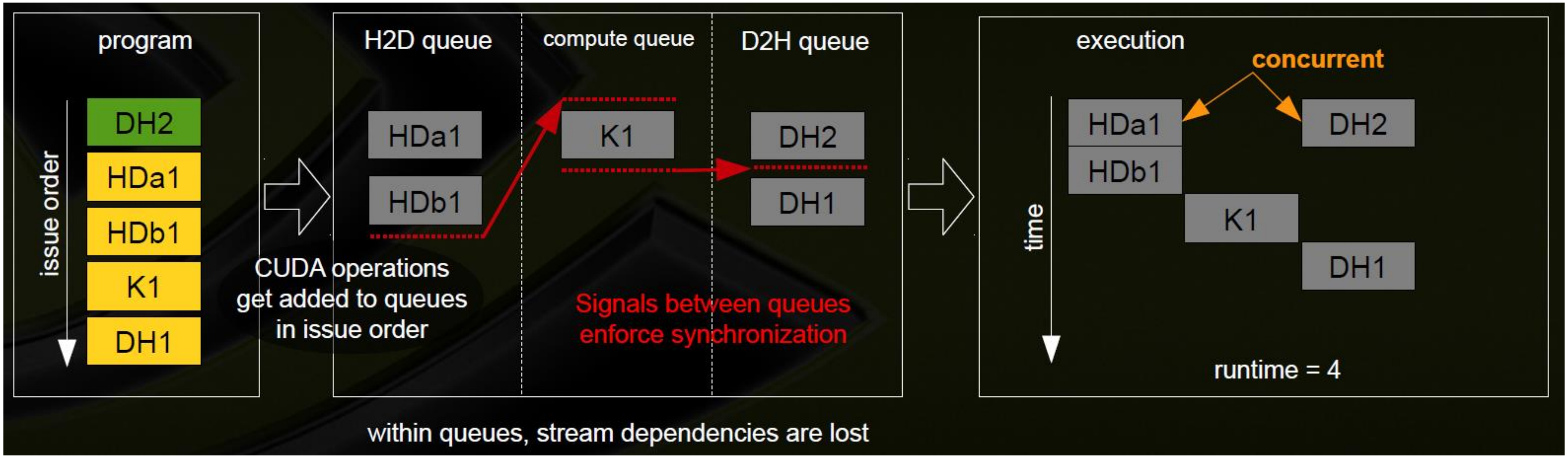
Example - Blocked Queue

- Two streams, stream 1 is issued first
  - Stream 1 : HDa1, HDb1, K1, DH1 (issued first)
  - Stream 2 : DH2 (completely independent of stream 1)



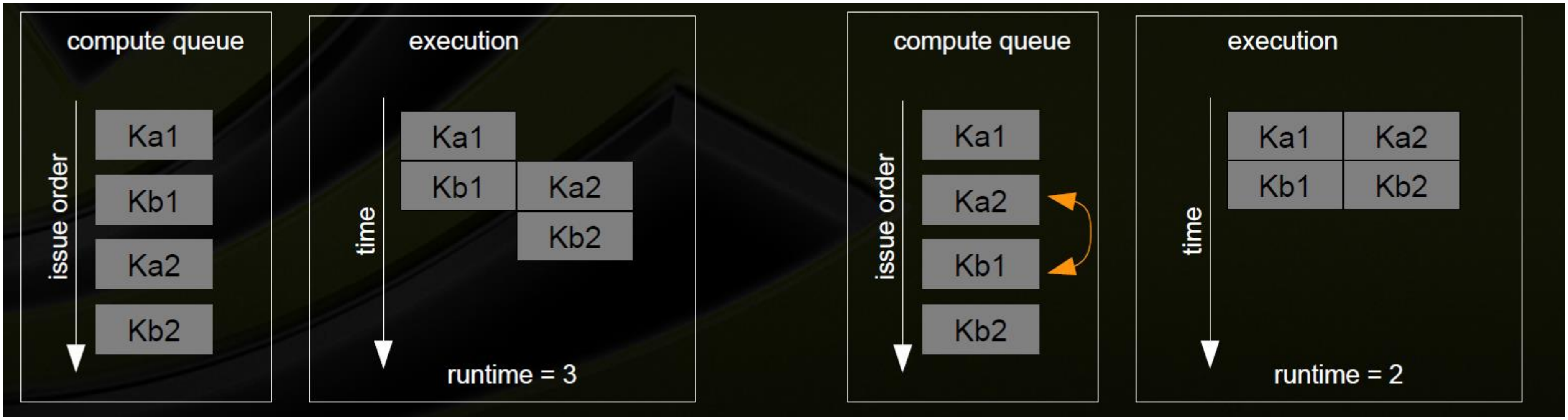
Example - Blocked Queue

- Two streams, stream 1 is issued first
  - Stream 1 : HDa1, HDb1, K1, DH1
  - Stream 2 : DH2 (issued first)



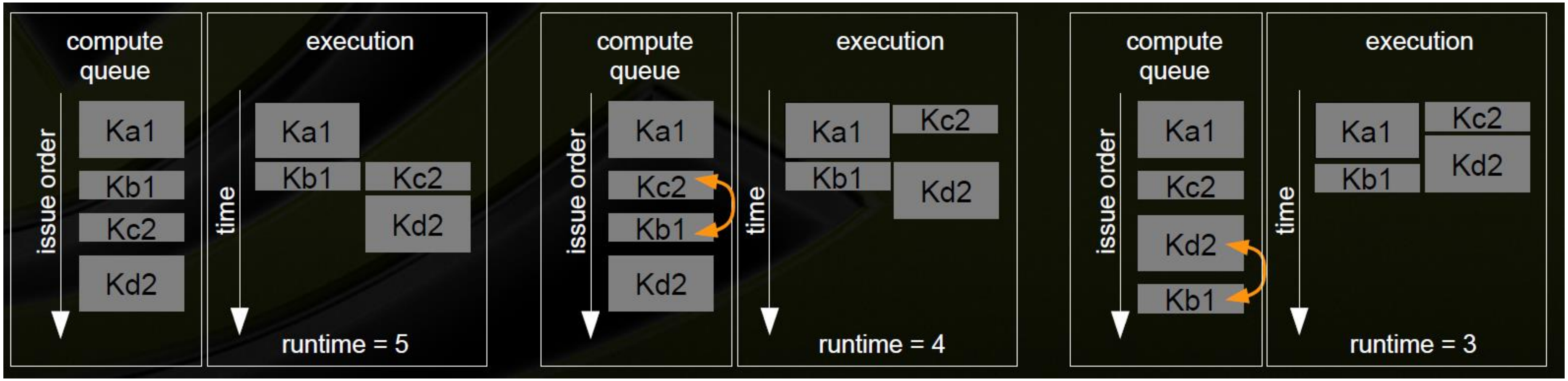
Example – Blocked Kernel

- Two streams – just issuing CUDA kernels
  - Stream 1 : Ka1, Kb1
  - Stream 2 : Ka2, Kb2
  - Kernels are similar size, fill ½ of the SM resources



Example - Optimal Concurrency can Depend on Kernel Execution Time

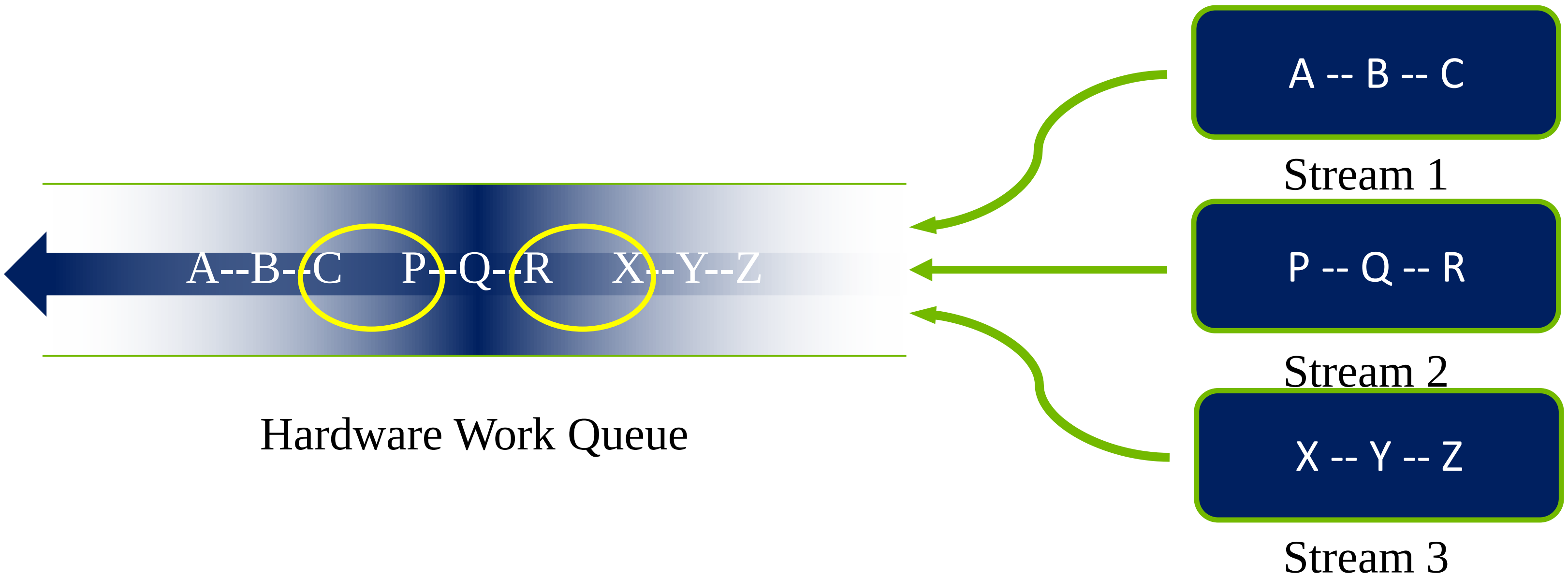
- Two streams - just issuing CUDA kernels - but kernels are different 'sizes'
  - Stream 1 : Ka1 {2}, Kb1 {1}
  - Stream 2 : Kc2 {1}, Kd2 {2}
  - Kernels fill ½ of the SM resources



Hyper Queue

- Provide multiple real queues for each engine
- Allow much more concurrency by allowing some streams to make progress for an engine while others are blocked

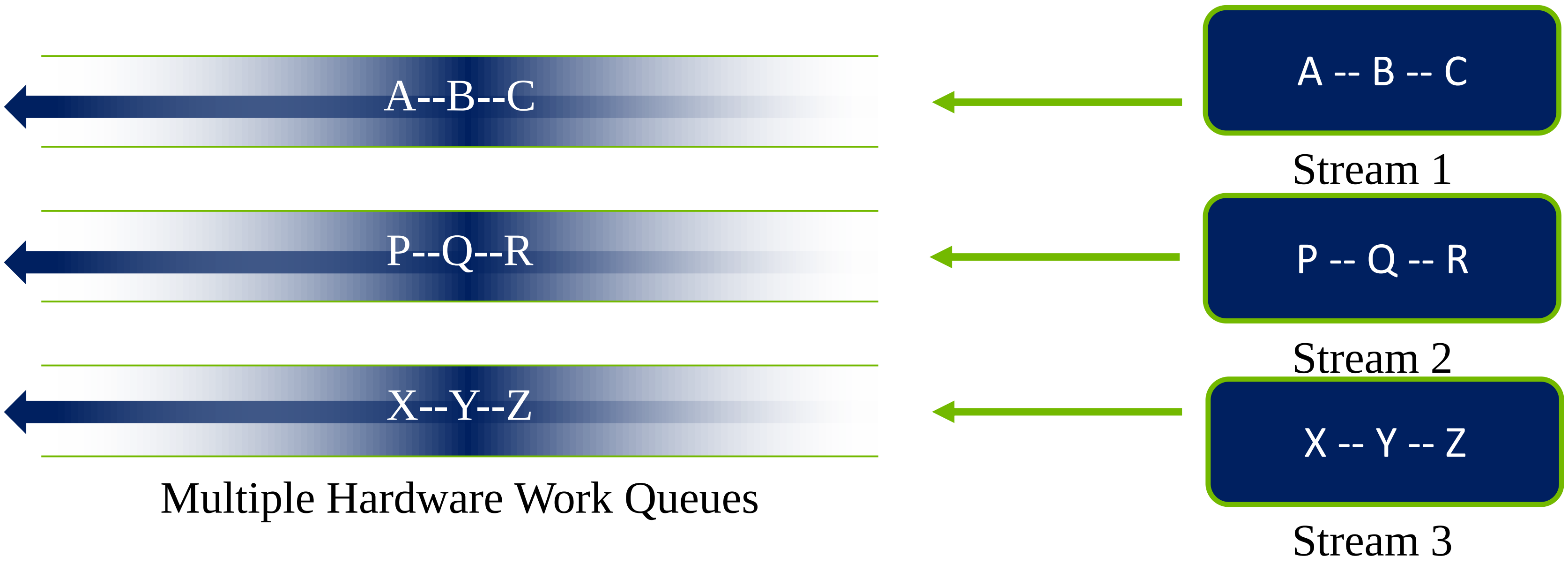
Fermi (and older) Concurrency



Fermi allows 16-way concurrency

– Up to 16 grids can run at once
– But CUDA streams multiplex into a single queue
– Overlap only at stream edges

Improved Concurrency



Kepler allows 32-way concurrency

- One work queue per stream
- Concurrency at full-stream level
- No inter-stream dependencies

Pascal also supports 32-way concurrency

- One work queue per stream
- Dynamic scheduling

Thrust

Jared Hoberock and Nathan Bell
Modified by P. Mordohai (March 2013)

NVIDIA

A Simple Example

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <iostream>

int main(void)
{
    // H has storage for 4 integers
    thrust::host_vector<int> H(4);

    // initialize individual elements
    H[0] = 14;
    H[1] = 20;
    H[2] = 38;
    H[3] = 46;
```

```
// H.size() returns the size of vector H
std::cout << "H has size " << H.size() << std::endl;

// print contents of H
for(int i = 0; i < H.size(); i++)
    std::cout << "H[" << i << "] = " << H[i] << std::endl;

// resize H
H.resize(2);

std::cout << "H now has size " << H.size() << std::endl;

// Copy host_vector H to device_vector D
thrust::device_vector<int> D = H;
```

```
    // elements of D can be modified
    D[0] = 99;
    D[1] = 88;

    // print contents of D
    for(int i = 0; i < D.size(); i++)
        std::cout << "D[" << i << "] = " << D[i] << std::endl;

    // H and D are automatically deleted when the function
  returns
    return 0;
}
```

Diving In

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void)
{
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;

    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());

    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
}
```

Objectives

- Programmer productivity
  - Rapidly develop complex applications
  - Leverage parallel primitives
- Encourage generic programming
  - Don't reinvent the wheel
  - E.g. one reduction to rule them all
- High performance
  - With minimal programmer effort
- Interoperability
  - Integrates with CUDA C/C++ code

What is Thrust?

- C++ template library for CUDA
  - Mimics Standard Template Library (STL)
- Containers
  - `thrust::host_vector<T>`
  - `thrust::device_vector<T>`
- Algorithms
  - `thrust::sort()`
  - `thrust::reduce()`
  - `thrust::inclusive_scan()`
  - Etc.

Namespaces

- C++ supports namespaces
  - Thrust uses `thrust` namespace
    - `thrust::device_vector`
    - `thrust::copy`
  - STL uses `std` namespace
    - `std::vector`
    - `std::list`
- Avoids collisions
  - `thrust::sort()`
  - `std::sort()`
- For brevity
  - `using namespace thrust;`

Containers

- Make common operations concise and readable
  - Hides `cudaMalloc`, `cudaMemcpy` and `cudaFree`

```
// allocate host vector with two elements
thrust::host_vector<int> h_vec(2);

// copy host vector to device
thrust::device_vector<int> d_vec = h_vec;

// manipulate device values from the host
d_vec[0] = 13;
d_vec[1] = 27;

std::cout << "sum: " << d_vec[0] + d_vec[1] << std::endl;

// vector memory automatically released w/ free() or cudaFree()
```

Containers

- Compatible with STL containers
  - Eases integration
  - `vector`, `list`, `map`, ...

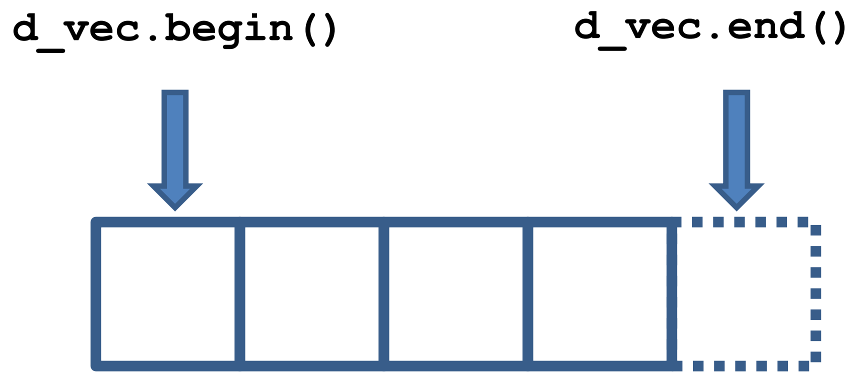
```
// list container on host
std::list<int> h_list;
h_list.push_back(13);
h_list.push_back(27);

// copy list to device vector
thrust::device_vector<int> d_vec(h_list.size());
thrust::copy(h_list.begin(), h_list.end(), d_vec.begin());

// alternative method
thrust::device_vector<int> d_vec(h_list.begin(), h_list.end());
```

Note: initializing an STL container with a device_vector works, but results in one cudaMemcpy() for each element instead of a single cudaMemcpy for the entire vector.

Iterators

- Sequences defined by pair of iterators

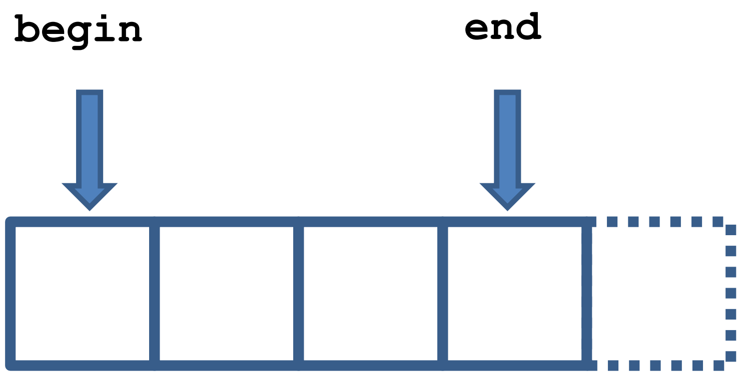
```
// allocate device vector
thrust::device_vector<int> d_vec(4);

d_vec.begin(); // returns iterator at first element of d_vec
d_vec.end()    // returns iterator one past the last element of d_vec

// [begin, end) pair defines a sequence of 4 elements
```



Iterators

- Iterators act like pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

thrust::device_vector<int>::iterator begin = d_vec.begin();
thrust::device_vector<int>::iterator end   = d_vec.end();

int length = end - begin; // compute size of sequence [begin, end)

end = d_vec.begin() + 3;  // define a sequence of 3 elements
```



Iterators

- Use iterators like pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

thrust::device_vector<int>::iterator begin = d_vec.begin();

*begin = 13;          // same as d_vec[0] = 13;
int temp = *begin;    // same as temp = d_vec[0];

begin++;              // advance iterator one position

*begin = 25;          // same as d_vec[1] = 25;
```

Iterators

- Track memory space (host/device)
  - Guides algorithm dispatch

```
// initialize random values on host
thrust::host_vector<int> h_vec(1000);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// copy values to device
thrust::device_vector<int> d_vec = h_vec;

// compute sum on host
int h_sum = thrust::reduce(h_vec.begin(), h_vec.end());

// compute sum on device
int d_sum = thrust::reduce(d_vec.begin(), d_vec.end());
```

Iterators

- Convertible to raw pointers

```
// allocate device vector
thrust::device_vector<int> d_vec(4);

// obtain raw pointer to device vector's memory
int * ptr = thrust::raw_pointer_cast(&d_vec[0]);

// use ptr in a CUDA C kernel
my_kernel<<<N/256, 256>>>(N, ptr);


// Note: ptr cannot be dereferenced on the host!
// raw pointers do not know where they live
// Thrust iterators do
```

Iterators

- Wrap raw pointers with `device_ptr`

```
int N = 10;

// raw pointer to device memory
int * raw_ptr;
cudaMalloc((void **) &raw_ptr, N * sizeof(int));

// wrap raw pointer with a device_ptr
thrust::device_ptr<int> dev_ptr(raw_ptr);

// use device_ptr in thrust algorithms
thrust::fill(dev_ptr, dev_ptr + N, (int) 0);

// access device memory through device_ptr
dev_ptr[0] = 1;

// extract raw pointer from device_ptr
int * raw_ptr2 = thrust::raw_pointer_cast(dev_ptr);

// free memory
cudaFree(raw_ptr);
```

Recap

- Containers
  - Manage host & device memory
  - Automatic allocation and deallocation
  - Simplify data transfers

- Iterators
  - Behave like pointers
  - Keep track of memory spaces
  - Convertible to raw pointers

- Namespaces
  - Avoid collisions

C++ Background

- Function templates

```
// function template to add numbers (type of T is variable)
template< typename T >
T add(T a, T b)
{
   return a + b;
}

// add integers
int x = 10; int y = 20; int z;
z = add<int>(x,y);    // type of T explicitly specified
z = add(x,y);         // type of T determined automatically

// add floats
float x = 10.0f; float y = 20.0f; float z;
z = add<float>(x,y);  // type of T explicitly specified
z = add(x,y);         // type of T determined automatically
```

C++ Background

- Function objects (Functors)

```
// templated functor to add numbers
template< typename T >
class add
{
    public:
    T operator()(T a, T b)
    {
       return a + b;
    }
};

int x = 10; int y = 20; int z;
add<int> func;    // create an add functor for T=int
z = func(x,y);    // invoke functor on x and y

float x = 10; float y = 20; float z;
add<float> func;  // create an add functor for T=float
z = func(x,y);    // invoke functor on x and y
```

```cpp
// this is a functor
// unlike functions, it can contain state
struct add_x {
  add_x(int x) : x(x) {}
  int operator()(int y) { return x + y; }

private:
  int x;
};

// Now you can use it like this:
add_x add42(42); // create an instance of the functor class
int i = add42(8); // and "call" it
assert(i == 50); // and it added 42 to its argument

std::vector<int> in; // assume this contains a bunch of values)
std::vector<int> out;
// Pass a functor to std::transform, which calls the functor on every
// element in the input sequence, and stores the result to the output
// sequence
// unlike a function pointer this can be resolved and inlined at
// compile time
std::transform(in.begin(), in.end(), out.begin(), add_x(1));
assert(out[i] == in[i] + 1); // for all i
```

C++ Background

- Generic Algorithms

```
// apply function f to sequences x, y and store result in z
template <typename T, typename Function>
void transform(int N, T * x, T * y, T * z, Function f)
{
   for (int i = 0; i < N; i++)
      z[i] = f(x[i], y[i]);
}

int N = 100;
int x[N]; int y[N]; int z[N];

add<int> func;                      // add functor for T=int

transform(N, x, y, z, func);        // compute z[i] = x[i] + y[i]

transform(N, x, y, z, add<int>()); // equivalent
```

Algorithms

- Thrust provides many standard algorithms
  - Transformations
  - Reductions
  - Prefix Sums
  - Sorting

- Generic definitions
  - General Types
    - Built-in types (`int`, `float`, …)
    - User-defined structures
  - General Operators
    - reduce with `plus` operator
    - scan with `maximum` operator

Algorithms

- General types and operators

```
#include <thrust/reduce.h>

// declare storage
device_vector<int>   i_vec = ...
device_vector<float> f_vec = ...

// sum of integers (equivalent calls)
reduce(i_vec.begin(), i_vec.end());
reduce(i_vec.begin(), i_vec.end(),    0, plus<int>());

// sum of floats (equivalent calls)
reduce(f_vec.begin(), f_vec.end());
reduce(f_vec.begin(), f_vec.end(), 0.0f, plus<float>());

// maximum of integers
reduce(i_vec.begin(), i_vec.end(),    0, maximum<int>());
```

Initial value of sum

Algorithms

- General types and operators

```
struct negate_float2
{
   __host__ __device__
   float2 operator()(float2 a)
   {
       return make_float2(-a.x, -a.y);
   }
};

// declare storage
device_vector<float2> input  = ...
device_vector<float2> output = ...

// create functor
negate_float2 func;

// negate vectors
transform(input.begin(), input.end(), output.begin(), func);
```

Algorithms

- General types and operators

```
// compare x component of two float2 structures
struct compare_float2
{
   __host__ __device__
   bool operator()(float2 a, float2 b)
   {
       return a.x < b.x;
   }
};

// declare storage
device_vector<float2> vec = ...

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
```

Algorithms

- Operators with State

```
// compare x component of two float2 structures
struct is_greater_than
{
   int threshold;

   is_greater_than(int t) { threshold = t; }

   __host__ __device__
   bool operator()(int x) { return x > threshold; }
};

device_vector<int> vec = ...

// create predicate functor (returns true for x > 10)
is_greater_than pred(10);

// count number of values > 10
int result = count_if(vec.begin(), vec.end(), pred);
```

Recap

- Algorithms
  - Generic
    - Support general types and operators
  - Statically dispatched based on iterator type
    - Memory space is known at compile time
  - Have default arguments
    - `reduce(begin, end)`
    - `reduce(begin, end, init, binary_op)`

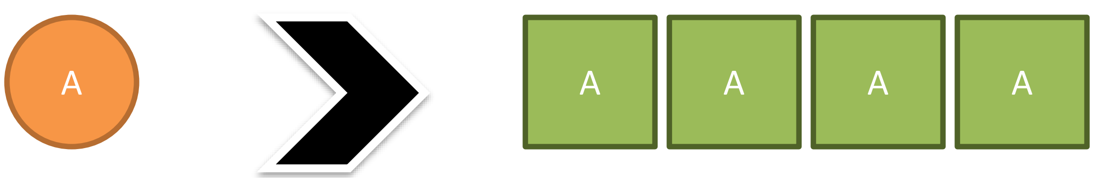
Fancy Iterators

- Behave like “normal” iterators
  - Algorithms don't know the difference
- Examples
  - `constant_iterator`
  - `counting_iterator`
  - `transform_iterator`
  - `permutation_iterator`
  - `zip_iterator`

Fancy Iterators

- `constant_iterator`
  - Mimics an infinite array filled with a constant value

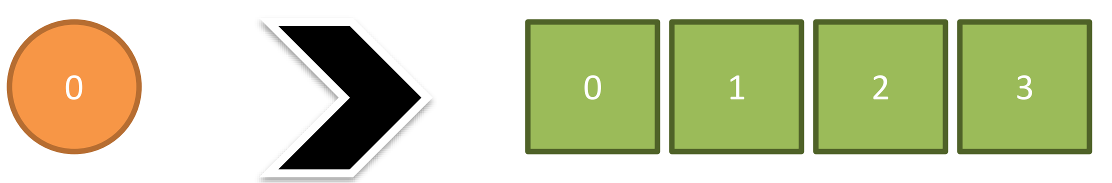
```
// create iterators
constant_iterator<int> begin(10);
constant_iterator<int> end = begin + 3;

begin[0]   // returns 10
begin[1]   // returns 10
begin[100] // returns 10

// sum of [begin, end)
reduce(begin, end);   // returns 30 (i.e. 3 * 10)
```



Fancy Iterators

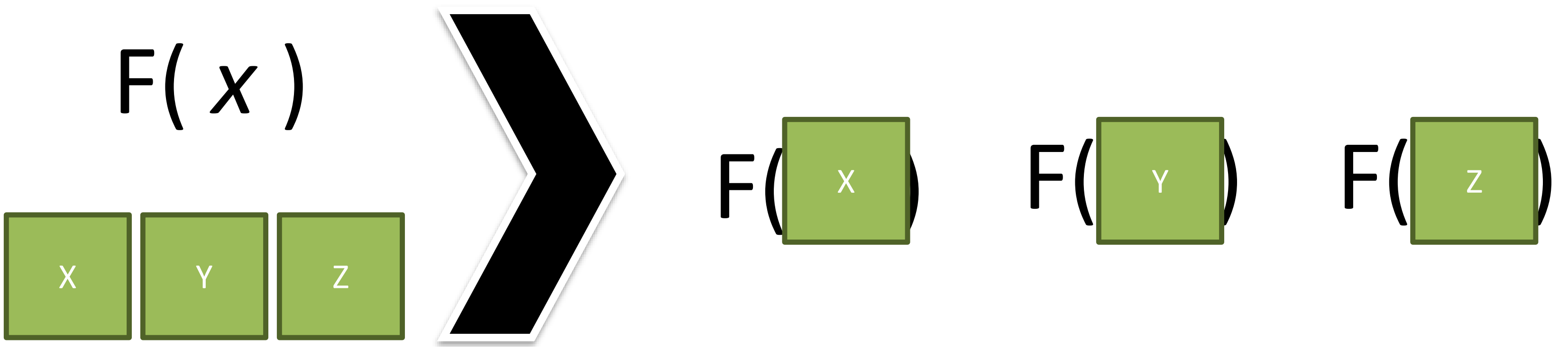
- `counting_iterator`
  - Mimics an infinite array with sequential values

```
// create iterators
counting_iterator<int> begin(10);
counting_iterator<int> end = begin + 3;

begin[0]   // returns 10
begin[1]   // returns 11
begin[100] // returns 110

// sum of [begin, end)
reduce(begin, end);   // returns 33 (i.e. 10 + 11 + 12)
```



Fancy Iterators

- `transform_iterator`
  - Yields a transformed sequence
  - Facilitates kernel fusion (e.g. sum of squares)



Fancy Iterators

- `transform_iterator`
  - Conserves memory capacity and bandwidth

```
// initialize vector
device_vector<int> vec(3);
vec[0] = 10; vec[1] = 20; vec[2] = 30;

// create iterator (type omitted)
first = make_transform_iterator(vec.begin(), negate<int>());
last  = make_transform_iterator(vec.end(),   negate<int>());

first[0]   // returns -10
first[1]   // returns -20
first[2]   // returns -30

// sum of [begin, end)
reduce(first, last);   // returns -60 (i.e. -10 + -20 + -30)
```

Fancy Iterators

- `zip_iterator`
  - Looks like an array of structs (AoS)
  - Stored in structure of arrays (SoA)

Fancy Iterators

- `zip_iterator`

```
// initialize vectors
device_vector<int>  A(3);
device_vector<char> B(3);
A[0] = 10;  A[1] = 20;  A[2] = 30;
B[0] = 'x'; B[1] = 'y'; B[2] = 'z';

// create iterator (type omitted)
first = make_zip_iterator(make_tuple(A.begin(), B.begin()));
last  = make_zip_iterator(make_tuple(A.end(),   B.end()));

first[0]   // returns tuple(10, 'x')
first[1]   // returns tuple(20, 'y')
first[2]   // returns tuple(30, 'z')

// maximum of [begin, end)
maximum< tuple<int,char> > binary_op;
reduce(first,last, first[0], binary_op); // returns tuple(30,'z')
// tuple() defines a comparison operator
```

Best Practices

- Fusion
  - Combine related operations together
- Structure of Arrays
  - Ensure memory coalescing
- Implicit Sequences
  - Eliminate memory accesses

Fusion

- Combine related operations together
  - Conserves memory bandwidth
- Example: SNRM2
  - Square each element
  - Compute sum of squares and take `sqrt()`
  - The fused implementation reads the array once while the un-fused implementation performs 2 reads and 1 write per element

Fusion

- Unoptimized implementation

```
// define transformation f(x) -> x^2
struct square
{
    __host__ __device__
    float operator()(float x)
    {
        return x * x;
    }
};

float snrm2_slow(device_vector<float>& x)
{
  // without fusion
  device_vector<float> temp(x.size());
  transform(x.begin(), x.end(), temp.begin(), square());

  return sqrt( reduce(temp.begin(), temp.end()) );
}
```

Fusion

- Optimized implementation (3.8x faster)

```
// define transformation f(x) -> x^2
struct square
{
    __host__ __device__
    float operator()(float x)
    {
        return x * x;
    }
};

float snrm2_fast(device_vector<float>& x)
{
  // with fusion
  return sqrt( transform_reduce(x.begin(), x.end(),
                                square(), 0.0f, plus<float>()));
}
```

Structure of Arrays (SoA)

- Array of Structures (AoS)
  - Often does not obey coalescing rules
    - `device_vector<float3>`
- Structure of Arrays (SoA)
  - Obeys coalescing rules
  - Components stored in separate arrays
    - `device_vector<float> x, y, z;`
- Example: Rotate 3d vectors
  - SoA is 2.8x faster

Array of Structures (AoS)

```
struct rotate_float3
{
  __host__ __device__
  float3 operator()(float3 v)
  {
    float x = v.x;
    float y = v.y;
    float z = v.z;

    float rx = 0.36f*x +  0.48f*y + -0.80f*z;
    float ry =-0.80f*x +  0.60f*y +  0.00f*z;
    float rz = 0.48f*x +  0.64f*y +  0.60f*z;

    return make_float3(rx, ry, rz);
  }
};

...

device_vector<float3> vec(N);

transform(vec.begin(), vec.end, vec.begin(), rotate_float3());
```

Structure of Arrays (SoA)

```
struct rotate_tuple
{
  __host__ __device__
  tuple<float,float,float> operator()(tuple<float,float,float> v)
  {
    float x = get<0>(v);
    float y = get<1>(v);
    float z = get<2>(v);

    float rx = 0.36f*x +  0.48f*y + -0.80f*z;
    float ry =-0.80f*x +  0.60f*y +  0.00f*z;
    float rz = 0.48f*x +  0.64f*y +  0.60f*z;

    return make_tuple(rx, ry, rz);
  }
};

...

device_vector<float> x(N), y(N), z(N);

transform(make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
          make_zip_iterator(make_tuple(x.end(),   y.end(),   z.end())),
          make_zip_iterator(make_tuple(x.begin(), y.begin(), z.begin())),
          rotate_tuple());
```

Implicit Sequences

- Avoid storing sequences explicitly
  - Constant sequences
    - `[1, 1, 1, 1, … ]`
  - Incrementing sequences
    - `[0, 1, 2, 3, … ]`

- Implicit sequences require no storage
  - `constant_iterator`
  - `counting_iterator`

- Example
  - Index of the smallest element

Implicit Sequences

```
// return the smaller of two tuples
struct smaller_tuple
{
  tuple<float,int> operator()(tuple<float,int> a, tuple<float,int> b)
  {
    if (a < b)
      return a;
    else
      return b;
  }
};

int min_index(device_vector<float>& vec)
{
  // create explicit index sequence [0, 1, 2, ... )
  device_vector<int> indices(vec.size());
  sequence(indices.begin(), indices.end());

  tuple<float,int> init(vec[0],0);
  tuple<float,int> smallest;

  smallest = reduce(make_zip_iterator(make_tuple(vec.begin(), indices.begin())),
                    make_zip_iterator(make_tuple(vec.end(),   indices.end())),
                    init,
                    smaller_tuple());

  return get<1>(smallest);
}
```

Implicit Sequences

```
// return the smaller of two tuples
struct smaller_tuple
{
  tuple<float,int> operator()(tuple<float,int> a, tuple<float,int> b)
  {
    if (a < b)
      return a;
    else
      return b;
  }
};

int min_index(device_vector<float>& vec)
{
  // create implicit index sequence [0, 1, 2, ... )
  counting_iterator<int> begin(0);
  counting_iterator<int> end(vec.size());

  tuple<float,int> init(vec[0],0);
  tuple<float,int> smallest;

  smallest = reduce(make_zip_iterator(make_tuple(vec.begin(), begin)),
                    make_zip_iterator(make_tuple(vec.end(),     end)),
                    init,
                    smaller_tuple());

  return get<1>(smallest);
}
```

Recap

- Best Practices
  - Fusion
    - 3.8x faster
  - Structure of Arrays
    - 2.8x faster
  - Implicit Sequences
    - 3.4x faster

Additional Resources

- Thrust
  - Homepage http://thrust.github.io/
  - More
    http://docs.nvidia.com/cuda/thrust/index.html
    https://developer.nvidia.com/thrust