Asynchronous Operations & Dynamic Parallelism in CUDA

S7705 – Session 3 of 4

Chris Mason
Product Manager, Acceleware
GPU Technology Conference
Date: May 8, 2017

About Acceleware

Programmer Training

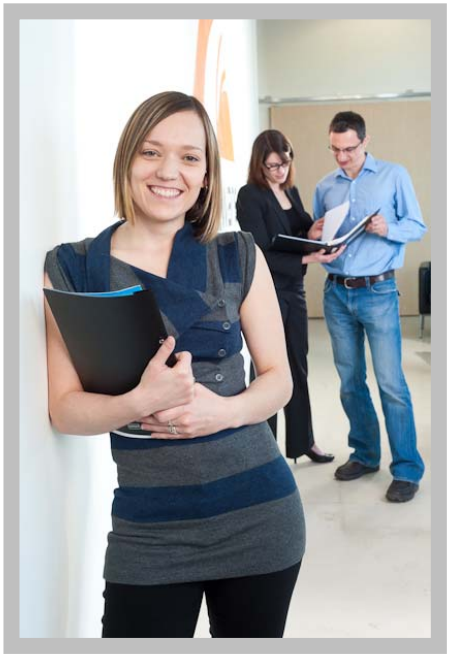
- CUDA and other HPC training classes
- Over 100 courses taught
- http://acceleware.com/training

Consulting Services

- Projects for Oil & Gas, Medical, Finance, Security and Defence, CAD, Media & Entertainment
- Mentoring, code review and complete project implementation
- http://acceleware.com/services

GPU Accelerated Software

- Seismic imaging & modeling
- Electromagnetics



acceleware

Seismic Imaging & Modeling

AxWAVE™

- Seismic forward modeling
- 2D, 3D, constant and variable density models
- High fidelity finite-difference modeling

AxRTM™

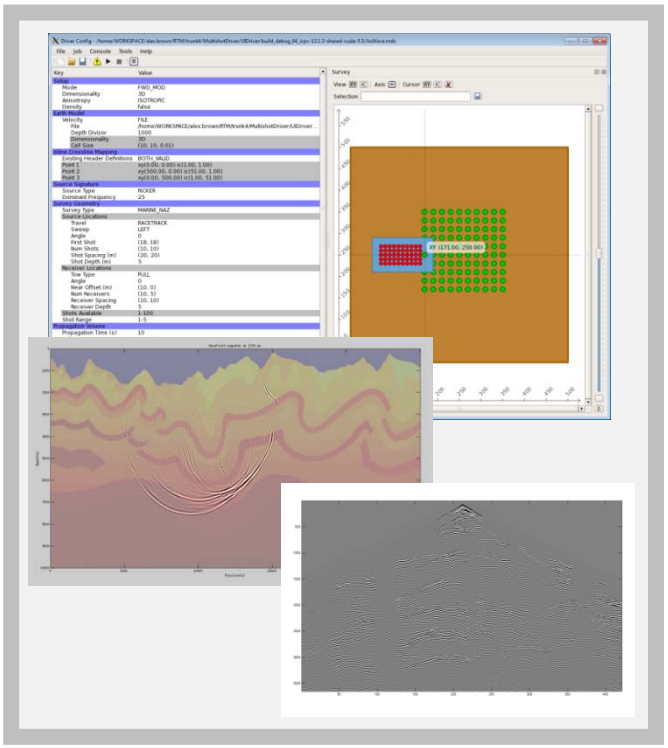
- High performance Reverse Time Migration application
- Isotropic, VTI and TTI media

AxFWI™

- Inversion of the full seismic data to provide an accurate subsurface velocity model
- Customizable for specific workflows

HPC Implementation

- Optimized for NVIDIA Tesla GPUs
- Efficient multi-GPU scaling

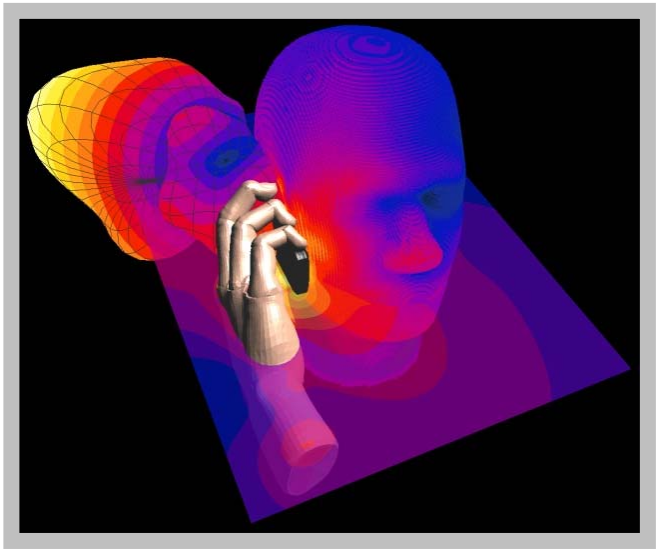


acceleware

Electromagnetics

AxFDTD™

- Finite-Difference Time-Domain Electromagnetic Solver
- Optimized for NVIDIA GPUs
- Sub-gridding and large feature coverage
- Multi-GPU, GPU clusters, GPU targeting



Available from:

acceleware

SYNOPSYS®

CROSSLIGHT

speag

Agilent Technologies

NLCSTR

Consulting Services

Industry	Application	Work Completed	Results
Finance	Option Pricing	Debugged & optimized existing CUDA code Implemented the Leisen-Reimer version of the binomial model for stock option pricing	30-50x performance improvement compared to single-threaded CPU code
Security & Defense	Detection System	Replaced legacy Cell-based infrastructure with GPUs Implemented a GPU accelerated X-ray iterative image reconstruction and explosive detection algorithms	Surpassed the performance targets Reduced hardware cost by a factor of 10
CAE	SIMULIA Abaqus	Developed a GPU accelerated version Conducted a finite-element analysis and developed a library to offload LDLT factorization portion of the multi-frontal solver to GPUs	Delivered an accelerated (2-3x) solution that supports NVIDIA and AMD GPUs
Medical	CT Reconstruction Software	Developed a GPU accelerated application for image reconstruction on CT scanners and implemented advanced features including job batch manager, filtering and bad pixel corrections	Accelerated back projection by 31x
Oil & Gas	Seismic Application	Converted MATLAB research code into a standalone application & improved performance via algorithmic optimizations	20-30x speedup

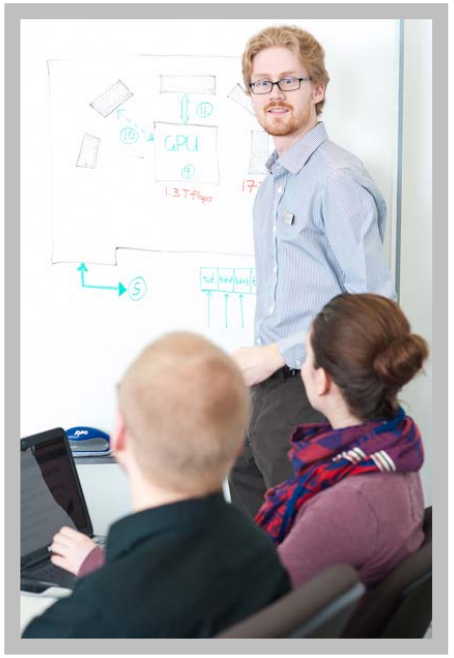
acceleware

Programmer Training

- CUDA and other HPC training classes
- Public, private onsite, and online courses
- Teachers with real world experience
- Hands-on lab exercises
- Progressive lectures
- Small class sizes to maximize learning
- 90 days post training support

"The level of detail is fantastic. The course did not focus on syntax but rather on how to expertly program for the GPU. I loved the course and I hope that we can get more of our team to take it."

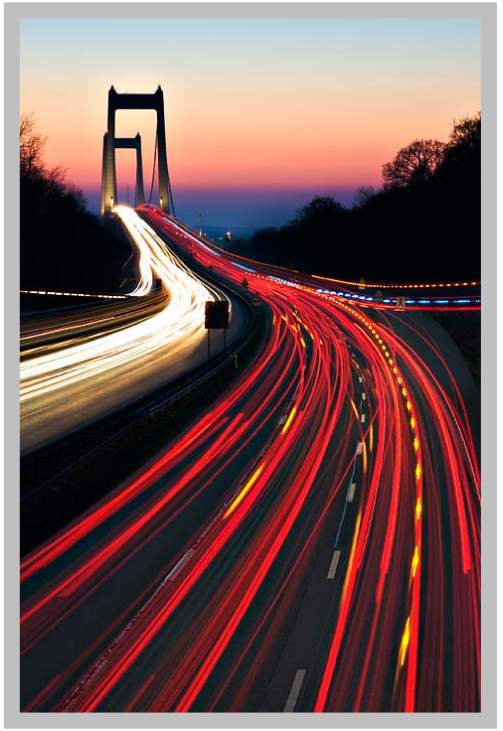
Jason Gauci, Software Engineer
Lockheed Martin



acceleware

Outline

- Concurrent CPU Computation and Kernel Execution
- Concurrent Transfers and Kernel Execution
- Streams/Events/Synchronization
- Pipelining
- Profilers and Asynchronous Operations
- Dynamic Parallelism



acceleware

Motivation

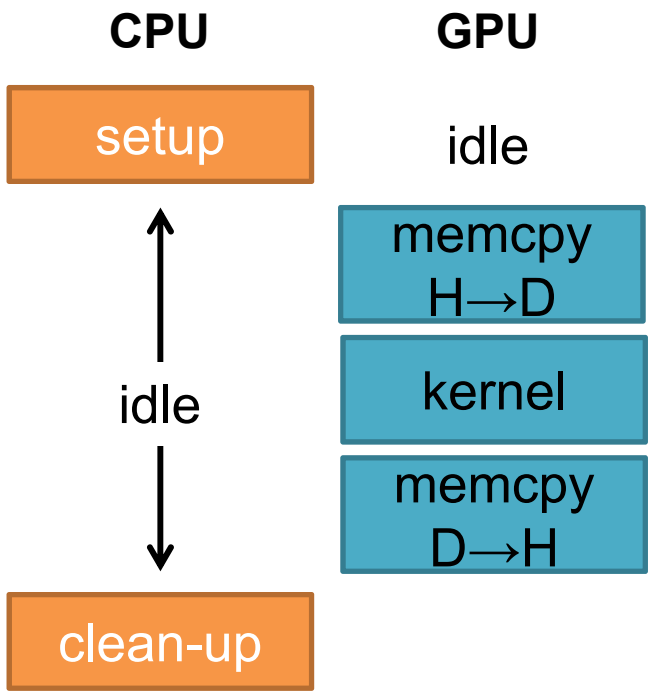
- Use asynchronous operations to introduce another level of parallelism
  - Better utilization of the system
  - Use all the available resources on your system concurrently
    - CPU cores
    - GPU(s)
    - PCIe bus
- Example:
  - Distribute a single large matrix multiply across multiple GPUs
  - Support matrices that are larger than available GPU memory by streaming data to/from host memory

Synchronicity in CUDA

- All CUDA calls are either synchronous or asynchronous relative to the host
  - Synchronous – enqueue work and wait for completion
  - Asynchronous – enqueue work and return immediately

What is the CPU doing during GPU kernels?

- Waiting for the GPU, typically…
- Use it for an independent task in parallel!
  - Eg. Reordering data, writing to a file



acceleware

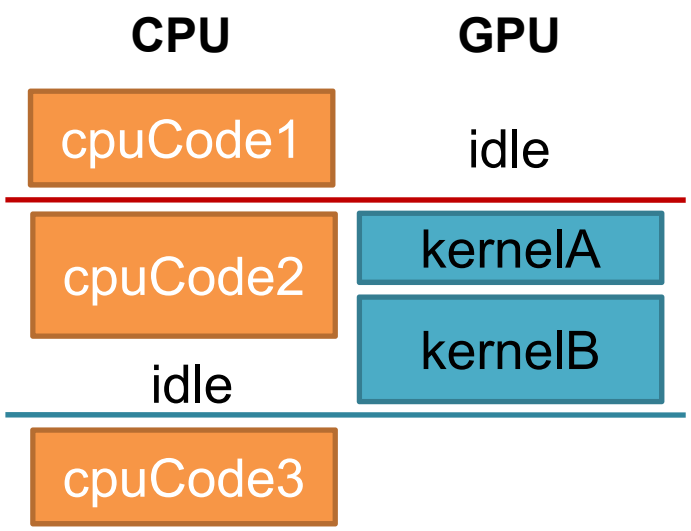
Concurrent CPU-GPU Computation

- Kernel invocations are asynchronous
- Need to synchronize CPU and GPU
  - Explicit: `cudaDeviceSynchronize()`
  - Implicit: `cudaMemcpy()`
- Order matters: launch kernels then call CPU functions

```
void foo()
{
  cpuCode1();

  KernelA<<<grid,block>>>(…);
  KernelB<<<grid,block>>>(…);


  cpuCode2();

  cudaDeviceSynchronize();

  // or cudaMemcpy for
  // implicit synchronization
  // cudaMemcpy(…,
  // cudaMemcpyDeviceToHost)
}
```


acceleware

Task Timeline

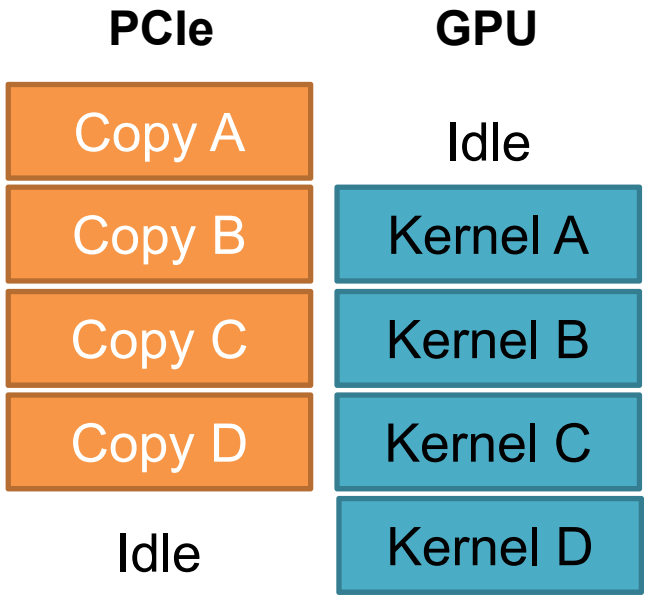



```
void foo()
{
  cpuCode1();

  KernelA<<<grid,block>>>(…);
  KernelB<<<grid,block>>>(…);

  cpuCode2();

  // Explicit synchronization
  cudaDeviceSynchronize();

  // Or implicit synchronization
  // cudaMemcpy(…)

  cpuCode3();
}
```


acceleware

Concurrent Transfers

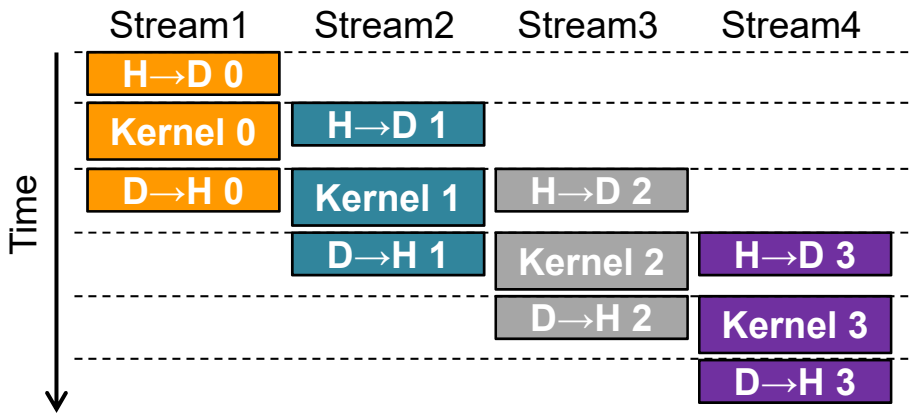
- Copy data over PCIe bus while kernels are executing on GPUS
- Check `asyncEngineCount` property in `cudaGetDeviceProperties()`
  - 0: No overlap is supported
  - 1: One data transfer can overlap with kernel execution
  - 2: Bidirectional data transfers can overlap with kernel execution
    - These devices have 2 DMA Engines!



acceleware

Asynchronous Communications

- Implemented via asynchronous variations of `cudaMemcpy*()`
- Stream = sequence (queue) of operations that execute *in order* on GPU
  - No ordering constraints on operations between different streams
- Operations can execute asynchronously if they:
  1) Are from different streams
  2) Use different hardware resources
  3) The necessary hardware resources are available



acceleware

Page-Locked Host Memory (1)

- Page-locked ("pinned") host memory required for asynchronous transfers
- `cudaMemcpyAsync()` does not require page-locked host memory
  - HOWEVER then transfers are synchronous
- `cudaMallocHost()` or `cudaHostAlloc()` and `cudaFreeHost()`
  - Allocate/free page-locked memory
- `cudaHostRegister()`/`cudaHostUnregister()`
  - Make existing memory page-locked

acceleware

Page-Locked Host Memory (2)

- Page-locked memory is typically faster with regular `cudaMemcpy()` too!
  - 2.7 vs 5.6GB/s on Tesla C2050, PCIe x16 Gen2
  - 3.3 vs 8.9GB/s on Tesla K40, PCIe x16 Gen 3
- Use with caution!
  - Allocating too much page-locked memory can reduce system performance
  - Calls to `cudaMallocHost()` will fail long before allocations by `malloc()`

Syntax – Asynchronous Communications

- Asynchronous Data Transfers
  - `cudaMemcpyAsync(void* dst, void* src, size_t nbytes, enum cudaMemcpyKind direction, cudaStream_t stream = 0)`
- Handles created and destroyed using API
  - `cudaError_t cudaStreamCreate(cudaStream_t* stream)`
  - `cudaError_t cudaStreamDestroy(cudaStream_t stream)`
- Launch kernels to different streams using the 4th argument in the chevron syntax <<<>>>
  - `myKernel<<<gSize, bSize, smBytes, stream>>>(…)`

acceleware

Default Stream

- Unless otherwise specified all operations are enqueued in the default stream or ‘’Stream 0”
- Default stream has special synchronization properties
  - Synchronous with all streams
    - Operations in default stream cannot overlap with operations from any other stream
- Exception: Streams with non-blocking flag set
  - `cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking)`
  - Allows operations in current stream to run concurrently with default stream
  - Use to get concurrency with operations out of your control (eg. libraries)

acceleware

Syntax Example

```
void foo(…)
{
  cudaStream_t stream1, stream2;

  cudaStreamCreate(&stream1);
  cudaStreamCreate(&stream2);

  cudaMemcpyAsync(d_bar, h_bar, sizeof(bar),
                  cudaMemcpyHostToDevice, stream1);

  kernelA<<<grid, block, 0, stream2>>>(…);
  kernelB<<<grid, block, 0, stream1>>>(…);

  …

  cudaStreamDestroy(stream1);
  cudaStreamDestroy(stream2);

}
```

These can overlap because they:

1) are on different streams
2) use different resources

- These are asynchronous with CPU execution
- Synchronization is required!

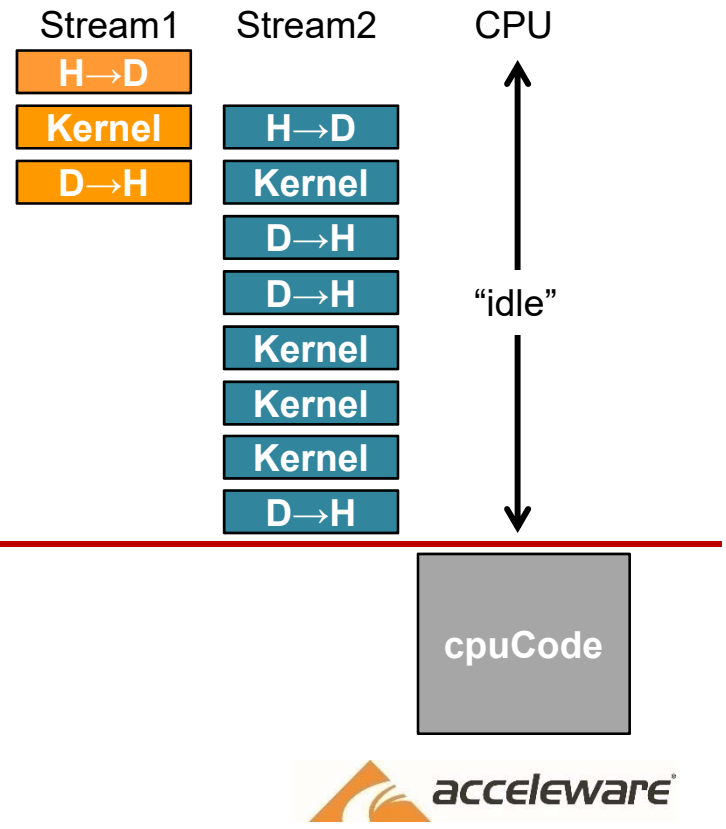
acceleware

Synchronization

1. `cudaDeviceSynchronize()`

CPU ↔ all operations on current device

```
cudaMemcpyAsync(…HostToDevice, stream1);

cudaMemcpyAsync(…HostToDevice, stream2);
Kernel<<<…, stream1>>>(…);

cudaMemcpyAsync(…DeviceToHost, stream1);
Kernel<<<…, stream2>>>(…);

cudaMemcpyAsync(…DeviceToHost, stream2);
cudaMemcpyAsync(…DeviceToHost, stream2);
Kernel<<<…, stream2>>>(…);
Kernel<<<…, stream2>>>(…);
Kernel<<<…, stream2>>>(…);
cudaMemcpyAsync(…DeviceToHost, stream2);

cudaDeviceSynchronize();
cpuCode();
```

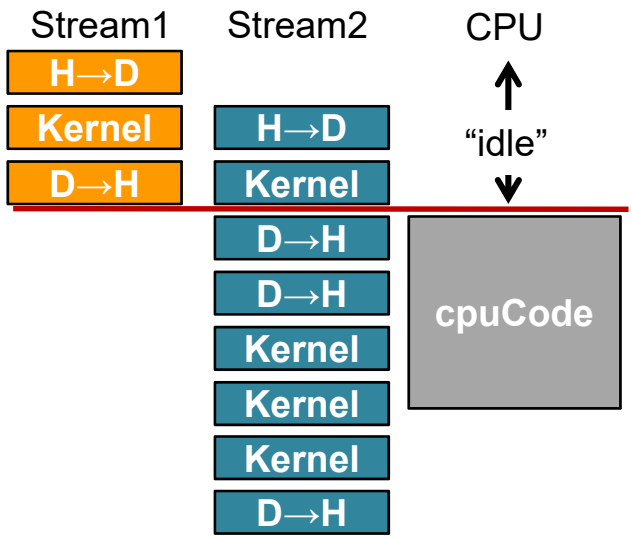


acceleware

Synchronization

2. `cudaStreamSynchronize()`
   - CPU ↔ all operations in a stream

```
cudaMemcpyAsync(…HostToDevice, stream1);

cudaMemcpyAsync(…HostToDevice, stream2);
Kernel<<<…, stream1>>>(…);

cudaMemcpyAsync(…DeviceToHost, stream1);
Kernel<<<…, stream2>>>(…);

cudaMemcpyAsync(…DeviceToHost, stream2);
cudaMemcpyAsync(…DeviceToHost, stream2);
Kernel<<<…, stream2>>>(…);
Kernel<<<…, stream2>>>(…);
Kernel<<<…, stream2>>>(…);
cudaMemcpyAsync(…DeviceToHost, stream2);

cudaStreamSynchronize(stream1);
cpuCode();
```



acceleware

Events (1)

- Events provide a mechanism to determine when operations in a stream are complete
  - ‘Dummy’ operations that do no work BUT
  - Useful for profiling and synchronization
- Events have a boolean state
  - Occurred (`cudaSuccess`)
  - Not Occurred
  - Default state = Occurred

acceleware

Events (2)

- Create and destroy events
  - `cudaEventCreate(cudaEvent_t* e)`
  - `cudaEventDestroy(cudaEvent_t e)`
- Record an event
  - `cudaEventRecord(cudaEvent_t e, cudaStream_t s = 0)`
    - Sets the event state to not occurred
    - Enqueues the event into a stream
    - Event state is set to occurred when all previous operations in the stream are complete
      - If stream is default stream, all prior operations in ALL other streams must be complete too!

acceleware

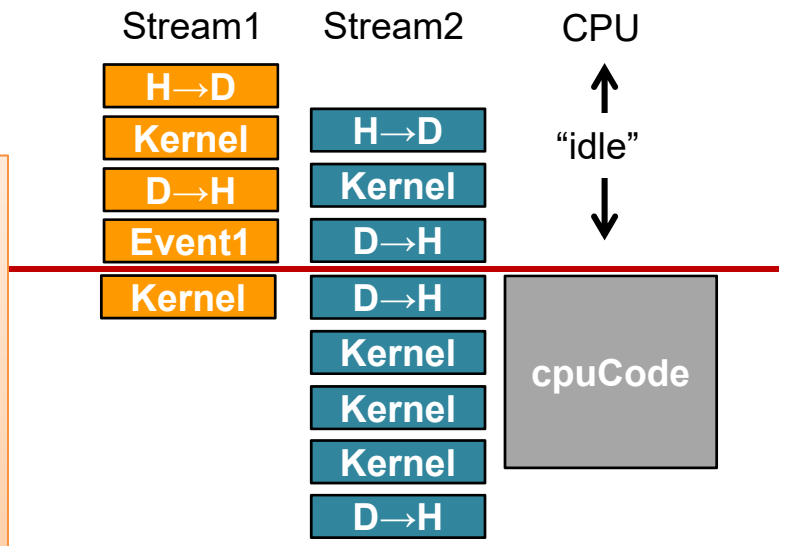
Event Synchronization

3. cudaEventSynchronize()

- CPU ↔ specific point in a stream's queue of operations

```
cudaMemcpyAsync(…HostToDevice, stream1);
Kernel<<<…, stream1>>>(…);
cudaMemcpyAsync(…DeviceToHost, stream1);
cudaEventRecord(event1,stream1);
Kernel<<<…, stream1>>>(…);

cudaMemcpyAsync(…HostToDevice, stream2);
Kernel<<<…, stream2>>>(…);
cudaMemcpyAsync(…DeviceToHost, stream2);
cudaMemcpyAsync(…DeviceToHost, stream2);
Kernel<<<…, stream2>>>(…);
Kernel<<<…, stream2>>>(…);
Kernel<<<…, stream2>>>(…);
cudaMemcpyAsync(…DeviceToHost, stream2);
```

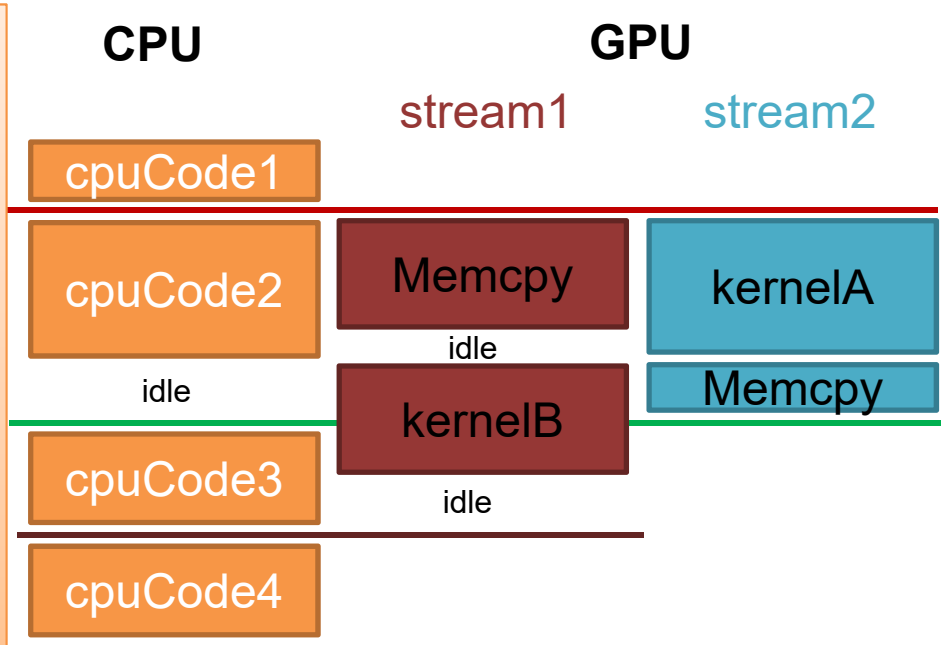
```
cudaEventSynchronize(event1);
cpuCode();
```



acceleware

Example: Putting it all Together

```
void foo()
{
  cpuCode1(…);

  cudaMemcpyAsync(…, stream1);

  kernelA<<<grid,block,0,stream2>>>(…);
  cudaMemcpyAsync(…, stream2);// Memcpy 2
  cudaEventRecord(event2, stream2);

  cpuCode2(…);

  kernelB<<<grid,block,0,stream1>>>(…);
  cudaEventRecord(event1, stream1);

  cudaEventSynchronize(event2);
  cpuCode3(…);

  cudaEventSynchronize(event1);
  cpuCode4(…);
}
```

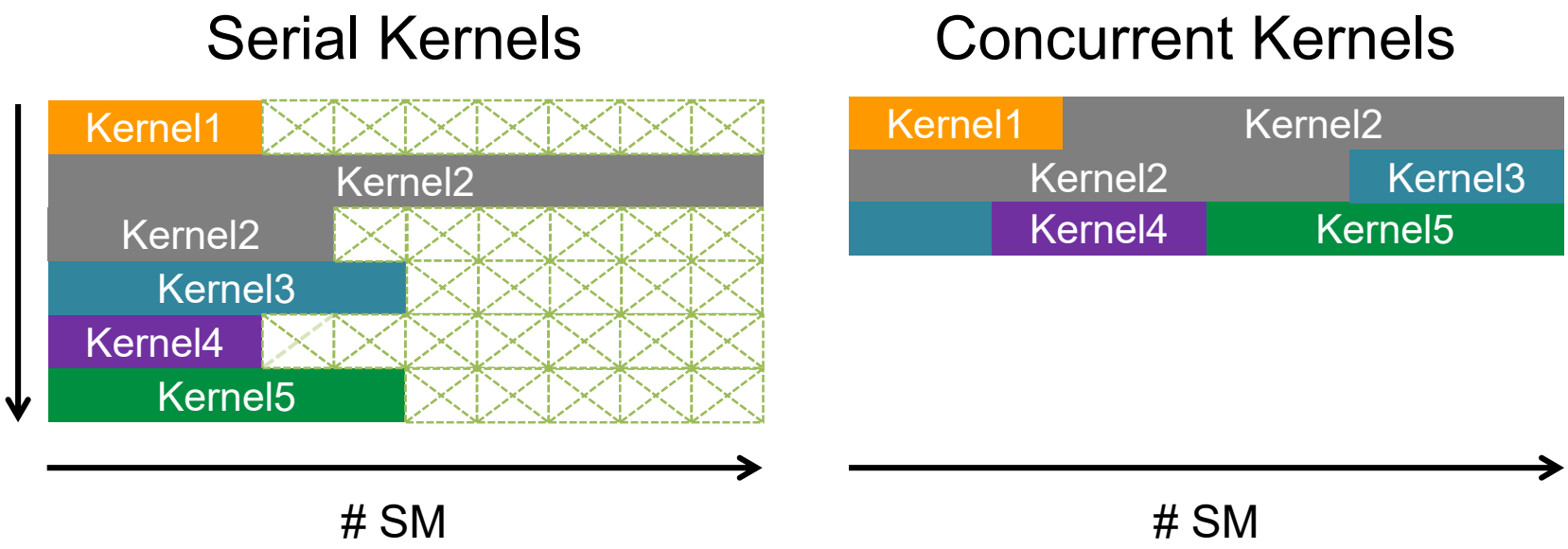


acceleware

Concurrent Kernel Execution

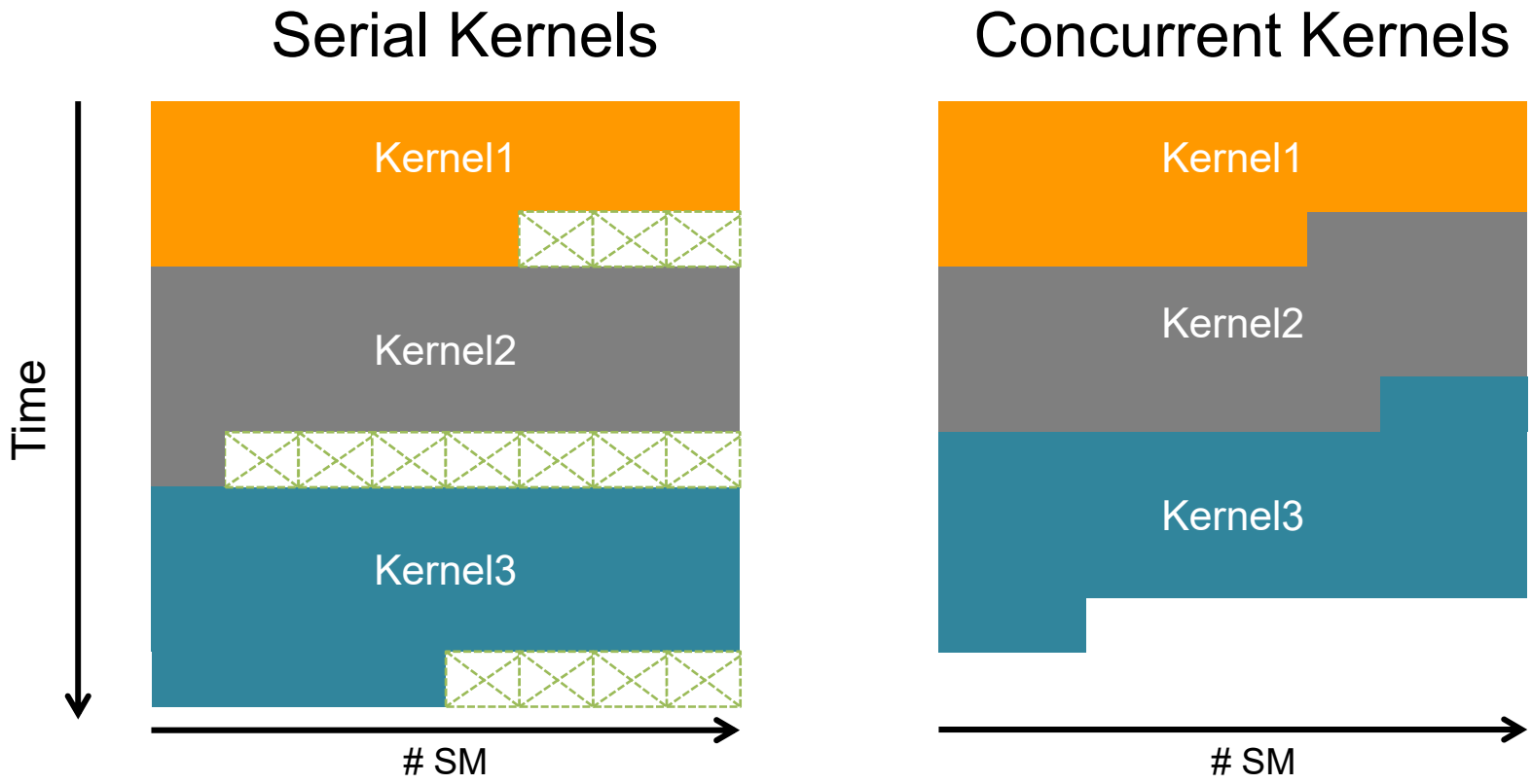
- Some Compute 2.0+ devices can execute multiple kernels concurrently
  - Check **concurrentKernels** property in **cudaGetDeviceProperties**
  - Maximum number of concurrent kernels
    - 32 kernels on compute capability 3.5 devices
    - 16 kernels on compute capability 2.0 – 3.0 devices
  - Kernels must be from the same process
  - Kernels must be from different, **non-default**, streams

Concurrent Kernel Execution



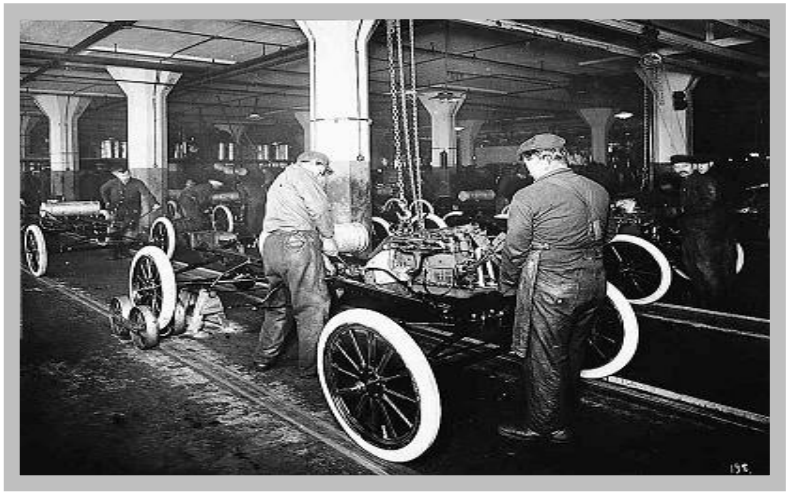
acceleware

Concurrent Kernel Execution



Pipelining

- Pipelining – a set of data processing elements connected in series
  - The output of one element is the input of the next
- Computing equivalent of assembly lines



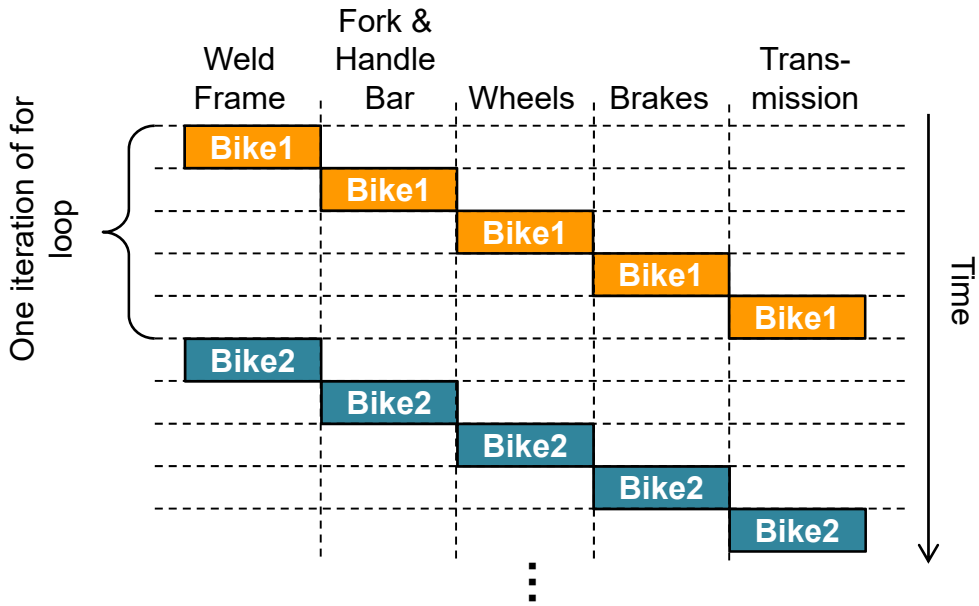
acceleware

Pipelining in CUDA

- Pipelining is common in CUDA as you have multiple resources, with each resource specialized for different tasks:
  - CPU – serial algorithms, I/O
  - GPU SMs – parallel algorithms
  - GPU DMA engines – CPU→GPU transfers

Implementing A Software Pipeline

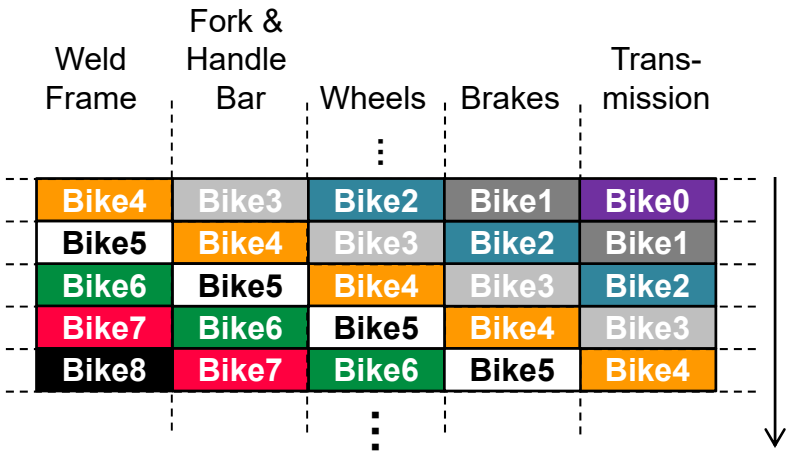
```
for(int bike = 0;
     bike < BICYCLES;
    ++bike)
{
    WeldFrame(bike);
    AttachForkAndHandleBar(bike);
    InstallWheelsAndTires(bike);
    InstallBrakes(bike);
    InstallChainAndTransmission(bike);
}
```



acceleware

Implementing A Software Pipeline

```
for(int bike = 0;
      bike < BICYCLES;
    ++bike)
{
     WeldFrameAsync(bike+2);
     AttachForkAndHandleBarAsync(bike+1);
     InstallWheelsAndTiresAsync(bike);
     InstallBrakesAsync(bike-1);
     InstallChainAndTransmissionAsync(bike-2);
}
```

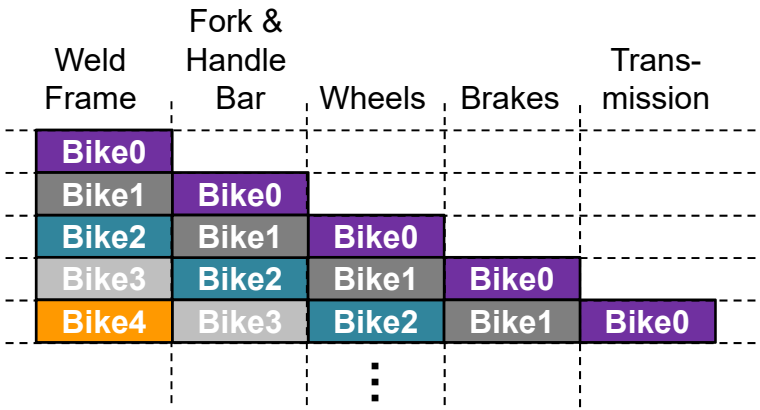


acceleware

Implementing a Software Pipeline

- One bicycle is completed every iteration, but 5 are under construction at the same time.
- Need to prime (and empty) the pipeline

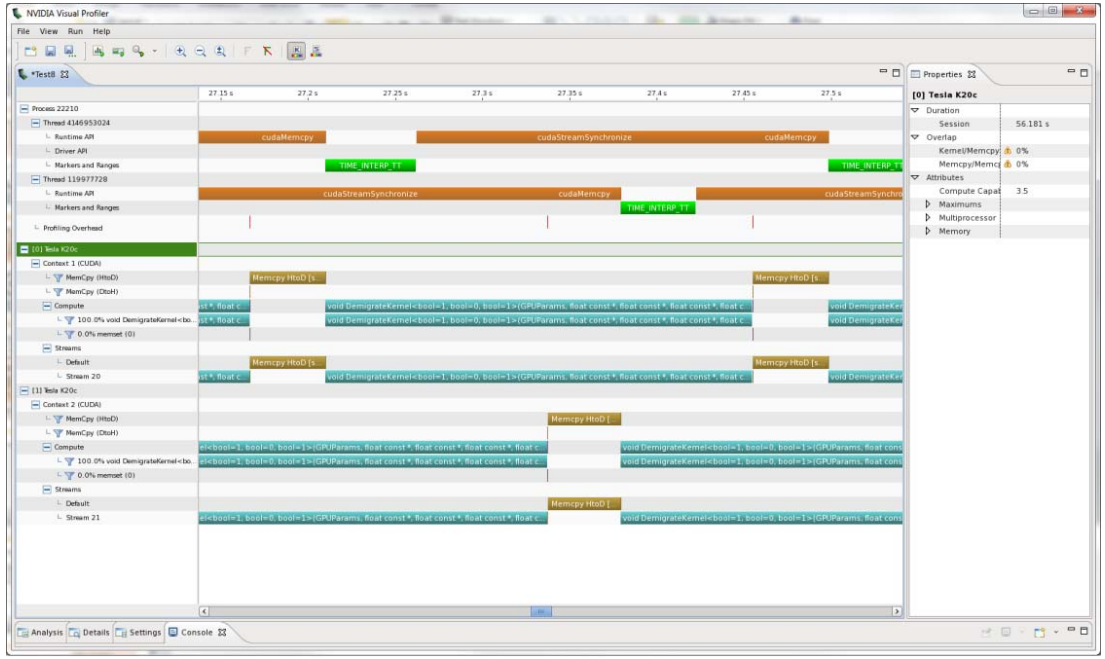
```
WeldFrame(0);

WeldFrame(1);
AttachForkAndHandleBar(0);

WeldFrame(2);
AttachForkAndHandleBar(1);
InstallWheelsAndTires(0);
…
for(int bike = 2; bike < BICYCLES-2;
   bike++)
```



acceleware

Profilers and Asynchronous Operations

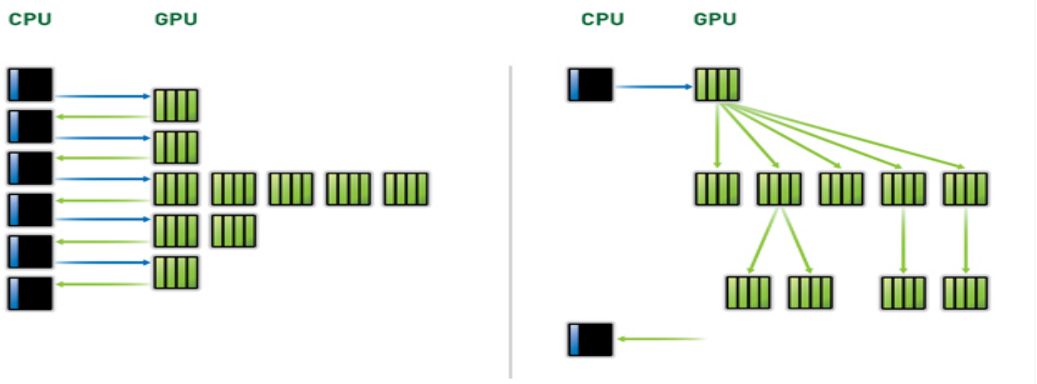
Timeline views useful for visualizing actual execution order of asynchronous operations



acceleware

Dynamic Parallelism

- Allows threads in GPU kernels to launch other GPU kernels
  - GPUs can operate more autonomously from host CPU



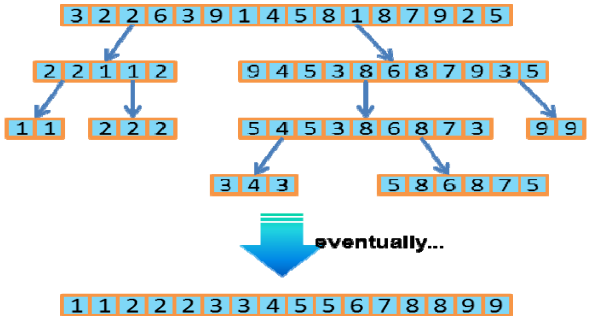
CPU Kernel Launches vs. Dynamic Parallelism

src: http://blogs.nvidia.com/blog/2012/09/12/how-tesla-k20-speeds-up-quicksort-a-familiar-comp-sci-code/

acceleware

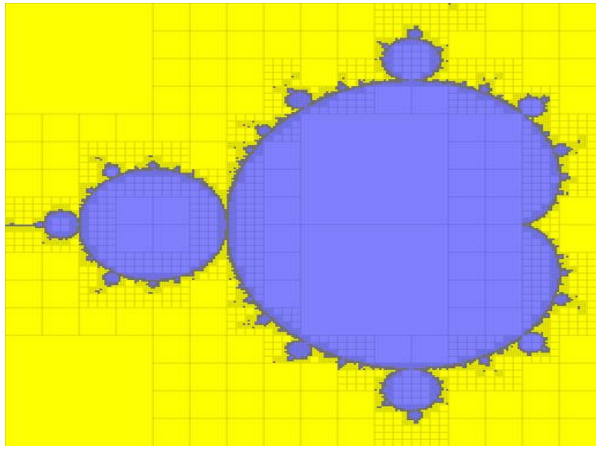
Dynamic Parallelism

- Dynamic Parallelism is well-suited to algorithms that use recursive subdivision



Quicksort Algorithm

src: http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/



Mariani-Silver Mandelbrot Set Algorithm

Dynamic Parallelism

- Consider Dynamic Parallelism for batch parallelism
  - Write a GPU kernel with 1 thread handling each batch of data
  - That thread can launch kernels for parallel computations on the batch

Dynamic Parallelism (Continued)

- Potential advantages of Dynamic Parallelism:
  - Simpler code for recursive problems
  - Eliminates device to host data transfers required for host to launch subsequent kernels
  - More efficient
    - Launch kernels from GPU as needed without any need for host synchronization that might introduce false data dependencies
      - eg. Non-Dynamic QuickSort implementation you need to wait for all kernels at one level of the tree are complete before launching any kernels at the next level!

Dynamic Parallelism

- Use <<<>>> syntax from within a kernel
- Synchronization
  - `cudaDeviceSynchronize()` synchronizes kernels launched from current thread block
  - All work launched by a thread block is implicitly synchronized when the block exits
    - Parent doesn't finish until children have finished

```
__global__ void childKernel()
{
    printf("Hello ");
}

__global__ void parentKernel()
{
    cudaError_t e;
    childKernel<<<1,1>>>();
    cudaDeviceSynchronize();
    e = cudaGetLastError();
    if(e) return;

    printf("World!\n");
}

void host_code()
{
    parentKernel<<<1,1>>>();
    ...
}
```


acceleware

Dynamic Parallelism & Memory Model

- Parent and child grids share the same global and constant memory storage
  - You can pass pointers to global/constant memory to child kernels
- Thread-private variables still have thread scope
  - Can't pass pointers to thread-private variables to child kernels
- Shared memory still has thread block scope
  - Can't pass pointers to shared memory to child kernels

```
__global__ void Child(int* );
__global__ void Child2(int );

__constant__ int constantX[10];

__global__ void ParentKernel(int *globalX)
{
    __shared__ int sharedX[10];
    int localX = 5;

    Child<<<1,1>>>(sharedX);          ✘
    Child<<<1,1>>>(&localX);          ✘
    Child2<<<1,1>>>(sharedX[0]);      ✓
    Child2<<<1,1>>>(localX);          ✓

    Child<<<1,1>>>(globalX);          ✓
    Child<<<1,1>>>(constantX);        ✓
}
```


acceleware

Dynamic Parallelism

- `cudaDeviceSynchronize()`
  - Parent blocks may pause execution and backup state (program counters, registers/shared memory contents) to host memory and yield to allow child kernels to make forward progress

acceleware

Dynamic Parallelism

- Kernels launched within a thread-block are executed in order by default
  - Use streams/events to allow concurrent execution of kernels
- Streams and Events are supported
  - A stream and event created by a thread has thread block scope
  - Using streams and events created on the host has undefined behavior
- Create Streams and Events using flags:
  - `cudaStreamCreateWithFlags()` with `cudaStreamNonBlocking` flag
  - `cudaEventCreateWithFlags()` with `cudaEventDisableTiming` flag

acceleware

Dynamic Parallelism & Streams

```
__global__ void parentKernel()
{
    cudaStream_t stream1, stream2;
    cudaStreamCreateWithFlags(&stream1, cudaStreamNonBlocking);
    cudaStreamCreateWithFlags(&stream2, cudaStreamNonBlocking);

    // launch child
    if(threadIdx.x == 0)
    {
        foo<<<1,1,0,stream1>>>();
    }
    else if(threadIdx.x == 32)
    {
        bar<<<1,1,0,stream2>>>();
    }
    ...
}
```


acceleware

Acceleware CUDA Training

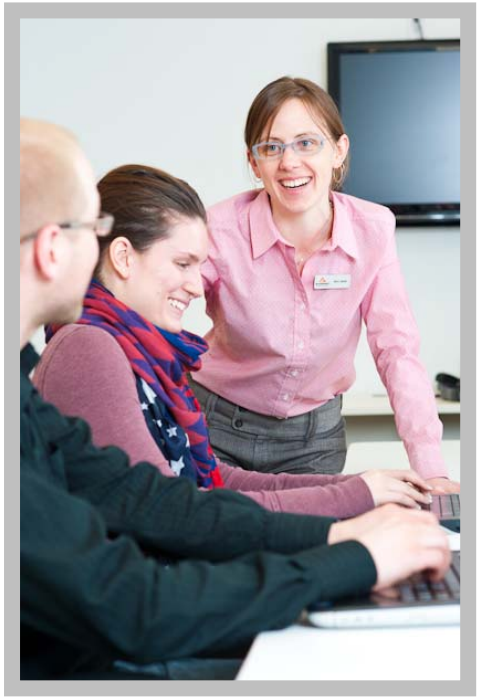
Scheduled CUDA Courses (also available online)

- June 13 – 16: Calgary, Alberta
  - 35% Discount using code: **AXECUDAGTC17**
- September 12 – 15: Calgary, Alberta
- December 5 – 8: Calgary, Alberta

Private training courses

- Courses held onsite at your company
- Delivered anywhere in the world

http://acceleware.com/cuda-training



acceleware

Questions?

Visit us at booth #520

Acceleware Ltd.
Tel: +1 403.249.9099
Email: **services@acceleware.com**

CUDA Blog: **http://acceleware.com/blog**
Website: **http://acceleware.com**

Chris Mason
chris.mason@acceleware.com