CUDA advanced aspects

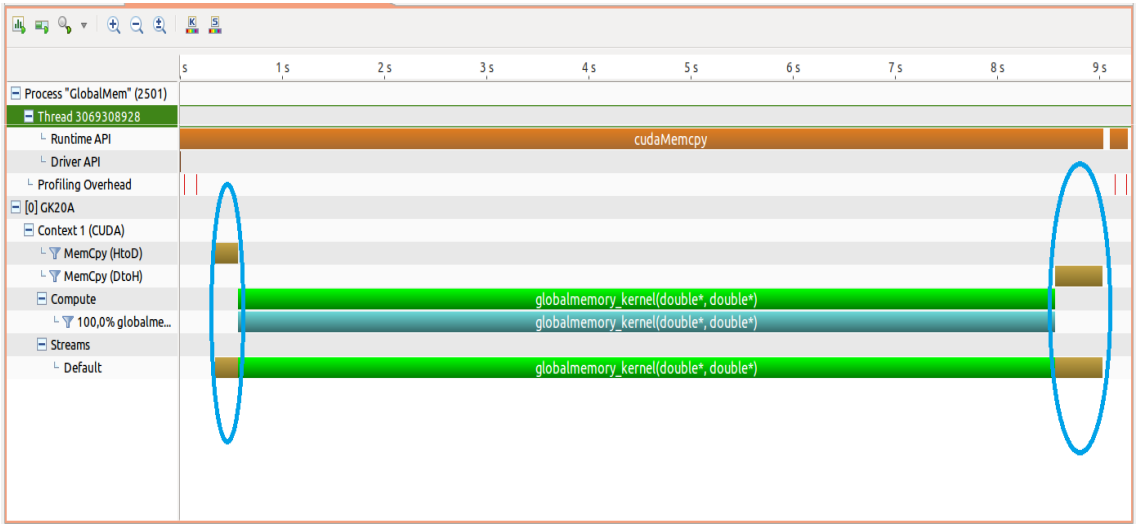
References

- CUDA C Programming Guide:
  - https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf
- CUDA C Best Practices Guide:
  - http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf
- David B. Kirk, When-mei W. Hwu – Programming Massively Parallel Processor, A Hands-on Approach (Second Edition)

Pinned Memory, Zero Copy, and Unified Memory

Loot at this situation…

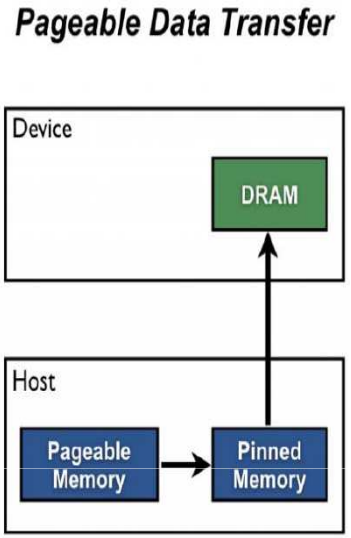
- Memory copy of 4MB
  - HostToDevice = **20 ms**
  - DeviceToHost = **40 ms**
- *Why*?

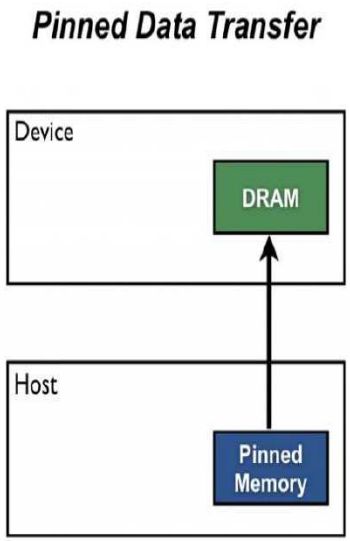


- In general, H2D and D2H transfers **can be expensive**
  - try to keep the computation local within the GPU, even for portions of code having similar execution times as the Host

Pageable vs. Pinned Memory

- *Answer*: The Host Memory is pageable and the Operating System can swap it!
  - Host data allocations are pageable by default
  - Operating System can swap them out to the disk
  - The DMA transfers to/from GPU cannot access data directly from pageable host memory
  - data transfers with the GPU first require the host data to be temporarily allocated as *page-locked*, or *pinned*, memory
  - You can explicitly request permanently **Pinned Memory** allocation of data
  - Use it with moderation, can decrease Host performance





Pinned Memory

- Using page-locked host memory has several benefits:
  - Copies between page-locked host memory and device memory can be performed concurrently with kernel execution for some devices (**Asynchronous Concurrent Execution**)
  - On some devices, page-locked host memory can be mapped into the address space of the device, eliminating the need to copy it to/from device memory (**Mapped Memory**)
  - Bandwidth between host memory and device memory can be higher if it is allocated as write-combining (**Write-Combining Memory**)
    - relies on a *write combine buffer* (WCB) allowing data to be combined and temporarily stored before being written in burst mode
    - doesn't guarantee write ordering but improves performance

Pinned Memory

- Use **`cudaHostAlloc()`** instead of **`malloc()`**

  `cudaErrot_t cudaHostAlloc(void** pHost, size_t size, unsigned int flags)`

- The **`flags`** parameter specifies how the allocation takes place:
  - **`cudaHostAllocDefault`**: This value (defined to be 0) causes **`cudaHostAlloc()`** to emulate **`cudaMallocHost()`**
  - **`cudaHostAllocPortable`**: The allocated memory will be pinned for all CUDA contexts, not only the one that performed the allocation
  - **`cudaHostAllocMapped`**: Maps the allocation into the CUDA address space. The device pointer to the memory may be obtained by calling **`cudaHostGetDevicePointer()`**
  - **`cudaHostAllocWriteCombined`**: Allocates the memory as *write-combined* (WC). WC memory can be transferred across the PCI Express bus more quickly on some system configurations, but cannot be read efficiently by most CPUs (a good option for buffers that will be written by the CPU and read by the device via mapped pinned memory or *host→device* transfers)

Zero Copy

- A block of page-locked host memory can also be mapped into the address space of the device
  - allows the kernel to *access host memory directly* from the GPU
  - use flag **`cudaHostAllocMapped`** in **`cudaHostAlloc()`**
- A block has two addresses (pointers):
  - one in host memory, returned by **`cudaHostAlloc()`** or **`malloc()`**
  - one in device memory, retrieved by **`cudaHostGetDevicePointer()`** and then used to access the block from the kernel
  - To do so, page-locked memory mapping must be enabled by calling **`cudaSetDeviceFlags()`** with the **`cudaDeviceMapHost`** flag before any other CUDA call is performed
  - Otherwise, **`cudaHostGetDevicePointer()`** will return an error

Zero Copy: example

```
int  *a,*b,*c;                 // host pointers
int *dev_a, *dev_b, *dev_c;    // device pointers to host memory

. . .
// determine device-side pointers to zero-copy data
cudaHostGetDevicePointer(&dev_a, a, 0);
cudaHostGetDevicePointer(&dev_b, b, 0);
cudaHostGetDevicePointer(&dev_c ,c, 0);

. . .

// Kernel launch
//  dev_a, dev_b, dev_c are passed to the kernel
//  The kernel will access host memory directly

add<<<B,T>>>(dev_a,dev_b,dev_c);
```

Zero Copy: Pros and Cons

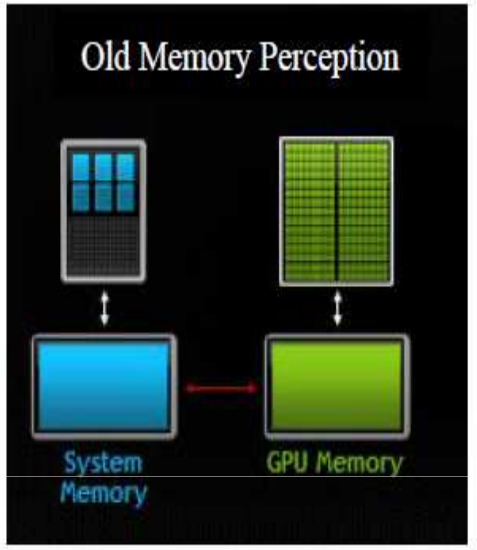
- Pros:
  - no need to allocate a block in device memory and copy data between this block and the block in host memory (*Zero Copy*)
    - data transfers are implicitly performed as needed by the kernel
  - There is no need to use streams to overlap data transfers with kernel execution
    - the kernel-originated data transfers automatically overlap with kernel execution.
- Cons:
  - Since memory is shared between host and device, application must synchronize accesses using streams or events to avoid any read-after-write, write-after-read, or write-after-write hazards
  - Atomic functions operating on mapped page-locked memory are not atomic from the point of view of the host or other devices

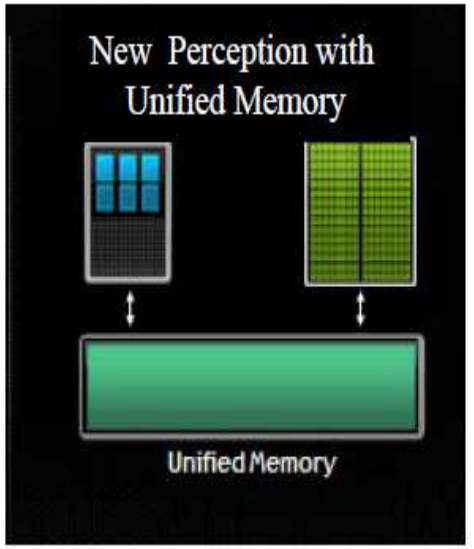
Unified Memory

- Managed memory space where all processors see a single, common, *coherent* address space
  - the underlying system manages data access and locality without need for explicit memory copy calls
  - First introduced in CUDA 6.0
- Simplifies GPU programming:
  - Unified Memory eliminates the need for *explicit data movement*
  - provides improved language integration for CUDA programmers
  - Enables writing simpler and more maintainable code
  - Data access speed is maximized by transparently migrating data towards the processor using it
  - no performance penalty incurred by placing all data into zero-copy memory

Unified Memory

- Offers a “single-pointer-to-data” model
  - conceptually similar to zero-copy memory
  - One key difference:
  - In zero-copy allocations the physical location of memory is pinned in CPU system memory (accesses from the GPU are slower)
  - Unified Memory: any allocation created in the managed memory space is automatically migrated to where it is used
- A program allocates managed memory in one of two ways:
  - **`cudaMallocManaged()`**
  - Defining a global **`__managed__`** variable





Unified Memory

- Managed memory migration is at the **page level**
  - The default page size is currently the same as the OS page size (typically **4 KB**)
- The runtime intercepts CPU dirty pages and detects page faults
  - Moves from device over PCIe only the dirty page
  - Transparently, pages touched by the CPU (GPU) are moved back to the device (host) when needed
- Coherence points are kernel launch and device/stream synchronizations
  - Notice: the same memory *cannot* be operated upon, at the same time, by the device and host

Unified Memory

- Issues related to *managed memory size*:
  - no “oversubscription” of the device memory
    - if there are several devices available, the maximum amount of managed memory that can be allocated is the *smallest* of the memories available on the devices
- Issues related to *transfer/execution overlap*:
  - Pages from managed allocations touched by CPU migrated back to GPU before any kernel launch
    - no kernel execution/data transfer overlap in that stream
    - Overlap possible with UM but just like before it requires multiple kernels in *separate streams*
      - Enabled by the fact that a managed allocation can be specific to a stream
      - Allows one to control which allocations are synchronized on specific kernel launches, enables concurrency

Unified Memory

- Issues related to *coherence*:
  - The GPU has exclusive access to this memory when any kernel is executed on the device
    - even if during its execution the kernel doesn't touch the managed memory
  - The CPU cannot access any managed memory allocation or variable as long as GPU is executing
  - A **`cudaDeviceSynchronize()`** call required for the host to be allowed to access managed memory
  - To this end, any function that logically guarantees the GPU finished execution is acceptable
    - Examples: **`cudaStreamSynchronize()`**, **`cudaMemcpy()`**, **`cudaMemset()`**, etc.

Unified Memory

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}

int main() {
    kernel << < 1, 1 >> >();
    y = 20; // ERROR: CPU access concurrent with GPU
    cudaDeviceSynchronize();
    return 0;
}
```

```
__device__ __managed__ int x, y = 2;
__global__ void kernel() {
    x = 10;
}

int main() {
    kernel << < 1, 1 >> >();
    cudaDeviceSynchronize();
    y = 20; // GPU is idle so access is OK
    return 0;
}
```

CUDA Asynchronous Concurrent Execution

Concurrent Host and Device execution

- In the CPU-GPU interplay, a CUDA enabled GPU can count on two “engines”
  - An execution engine
  - A copy engine, which actually has two subengines that can work simultaneously
    - A Host-to-Device (H2D) copy subengine
    - A Device-to-Host (D2H) copy subengine
- How to use both engines at the same time?

Asynchronous Concurrent Execution

- In order to facilitate concurrent execution on host and device, some function calls are ***asynchronous***
  - Control is returned to the host thread before the device has completed the requested task
- Examples of asynchronous calls:
  - Kernel launches
  - Device → device memory copies
  - Host → device memory copies of a block of 64 KB or less
  - Memory copies performed by functions suffixed with **…Async**
  - Note: When an application is run via a CUDA *debugger* or profiler (*cuda-gdb*, *nvvp*, *Parallel Nsight*), all launches are synchronous

Host-Device data transfer issues

- host → device data transfers using **`cudaMemcpy()`** are blocking
  - Control is returned to the host thread only *after* the data transfer is complete
- There is a non-blocking variant, **`cudaMemcpyAsync()`**

```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid,block>>>(a_d);
cpuFunction();
```

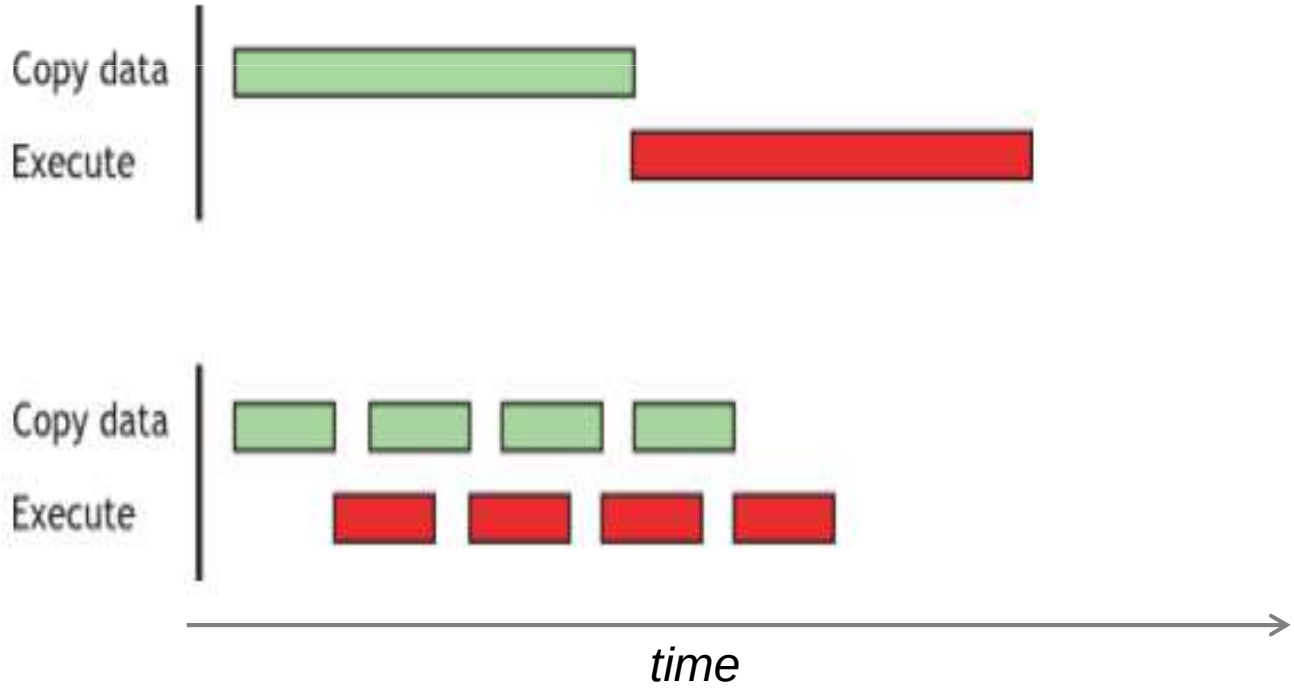
  - The host does not wait on the device to finish the memory copy and the kernel call for it to start execution of **`cpuFunction()`** call
  - The launch of “kernel” only happens after the memory copy call finishes
- Note 1: the asynchronous transfer version requires pinned host memory (allocated with **`cudaHostAlloc()`**), and it contains an additional argument (a stream ID)
- Note 2: we are still not using the two GPU engines concurrently
  - We only make the CPU stay busy

Overlapping device transfer and execution

- When is this overlapping useful?
  - Imagine a kernel executing on the device and only working with the lower half of the device global memory
  - Then, you can copy data from host to device into the upper half of the device global memory, which *is not currently being used*
  - These two operations can take place *simultaneously*
- Note that there is an issue with this idea:
  - The device execution stack is FIFO
    - one function call on the device is not serviced until all the previous device function calls completed
  - This would prevent overlapping execution with data transfer
- This issue is addressed by the use of CUDA “streams”

Overlapping device transfer and execution

- Communication and compute operations can take place simultaneously
- This is often referred to as *double buffering*



CUDA Streams: Overview

- A programmer can manage *concurrency* through ***streams***
  - “concurrency” refers to “the copy and the execution engines of the GPU working at the same time” or “multiple different kernels being executed at the same time on the GPU”
- A stream is a sequence of CUDA commands that execute in issue-order
  - Look at a stream as a queue of GPU operations
  - The execution order in a stream is identical to the order in which the GPU operations are added to the stream (**FIFO**)
  - Note: an operation in a stream does not start prior to the previous operation being fully completed
    - There is a distinction between queuing an operation in a stream and the moment when it actually starts to be executed on the GPU

CUDA Streams: Overview

- One host thread can define *multiple* CUDA streams
- What are the typical operations in a stream?
  - Invoking a data transfer
  - Invoking a kernel execution
  - Handling events
- How executions of different streams relate to each other?
  - Inter-stream relative behavior is ***not guaranteed*** and should therefore not be relied upon for correctness
  - e.g. inter-kernel communication for kernels allocated to different streams is undefined
  - Streams can be synchronized at barrier points, but correlation of sequence execution within different streams is not supported

CUDA Streams: Creation

- A stream is defined by creating a *stream* object
  - It is subsequently used by specifying it as the stream parameter to a sequence of kernel launches and *host* ↔ *device* memory copies
- The following code sample creates two streams and allocates an array "**`hostPtr`**" of float in page-locked memory
  - **`hostPtr`** will be used in asynchronous host ↔ device memory transfers

- Note: As soon you invoke a CUDA function you create a default stream (Stream 0)
  - If you don't explicitly state a stream in the execution configuration of a kernel it is assumed it is launched as part of "Stream 0"

```
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
        cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```

CUDA Streams: Using them

- In the code below, each of the two streams is defined as a sequence of
  - One memory copy from host to device,
  - One kernel launch, and
  - One memory copy from device to host

```
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);

    MyKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);

    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size,size, cudaMemcpyDeviceToHost, stream[i]);
}
```

CUDA Streams: Cleaning up

- Streams are released by calling **`cudaStreamDestroy()`**

```
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

- **`cudaStreamDestroy()`** waits for all preceding commands in the given stream to complete before destroying the stream and returning control to the host thread

CUDA Streams: Caveats

- Two commands from different streams cannot run concurrently if one of the following operations is issued in-between them by the host thread:
  - A page-locked host memory allocation,
  - A device memory allocation,
  - A device memory set,
  - A device ↔ device memory copy,
  - A CUDA command to Stream 0 (including kernel launches and host ↔ device memory copies that do not specify any stream parameter)
  - A switch between the L1/shared memory configurations

CUDA Streams: Caveats

- All GPU calls (**`memcpy`**, kernel execution, etc.) are placed into default stream unless otherwise specified

- Stream 0 is special
  - *Synchronous* with all streams
    - Meaning: Things done in Stream 0 *cannot overlap* with other streams
  - Streams with non-blocking flag are an exception:

```
cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking)
```

CUDA Streams: Synchronization

- **`cudaDeviceSynchronize()`** halts execution on the host until all preceding commands in all CUDA streams have completed
- **`cudaStreamSynchronize()`** takes a stream as a parameter and halts execution on the host until all preceding commands in the given CUDA stream have completed
  - can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device
- **`cudaStreamWaitEvent()`** takes a CUDA stream and an event as parameters and makes all the commands added to the given stream after the call to **`cudaStreamWaitEvent()`** delay their execution until the given event has completed
  - Note: this halts the execution of tasks on a stream
- **`cudaStreamQuery()`** provides applications with a way to know if all preceding commands in a stream have completed
- Note: To avoid unnecessary slowdowns, use these synchronization functions sparingly

Example: Use of **`cudaStreamWaitEvent`**

- Assume **`stream1`** and **`stream2`** have been defined/initialized already
- The point of this example:
  - Use *the two copy subengines at the same time*
  - Wait on the Stream 2 launching **`myKernel`** until the copy operation in Stream 1 is completed

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

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

cudaMemcpyAsync ( out, d_out, size, D2H, stream2 );        // 2) D2H copy of previous result

cudaStreamWaitEvent ( stream2, event );                    // wait for event in stream1
myKernel<<< 1000, 512, 0, stream2 >>> ( d_in, d_out );     // 3) GPU must wait for 1 and 2
someCPUfunction ( blah, blahblah )                         // this gets executed right away
```

Dynamic Parallelism

Additional references

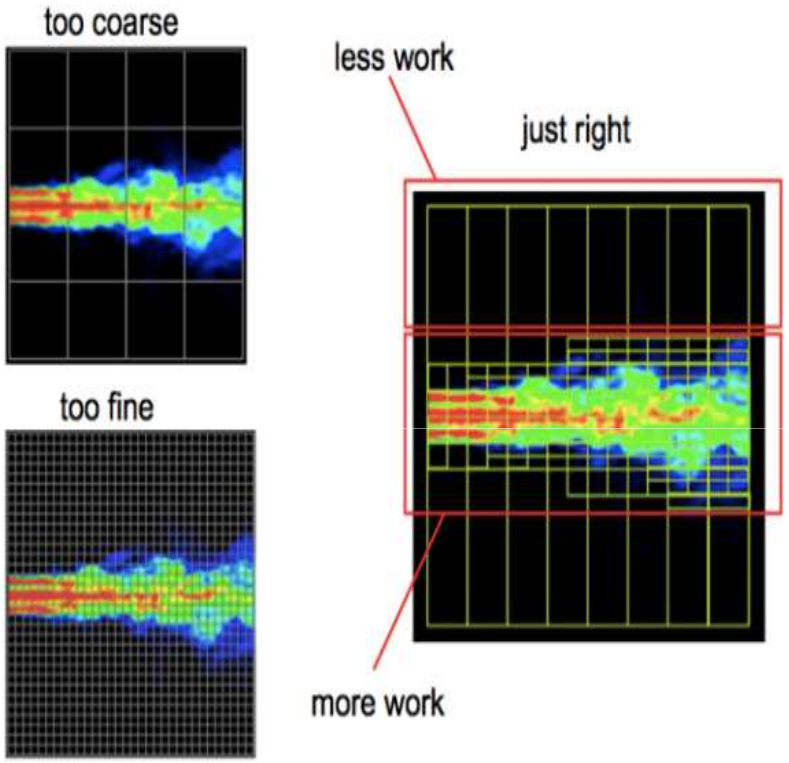
- CUDA Dynamic Parallelism API and Principles, PARALLEL FORALL
  - http://devblogs.nvidia.com/parallelforall/introduction-cuda-dynamic-parallelism/
  - http://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles/
  - http://devblogs.nvidia.com/parallelforall/a-cuda-dynamic-parallelism-case-study-panda/

Dynamic Parallelism: Why bother?

- Early CUDA programs had to conform to a flat, bulk parallel programming model
- Programs had to perform a sequence of kernel launches
  - for best performance each kernel had to expose enough parallelism to efficiently use the GPU
- But some parallel patterns (such as *nested* parallelism) cannot be expressed so easily with this model
  - flat, bulk parallel applications have to use either a fine grid, and do unwanted computations,…
  - or use a coarse grid and lose finer details

Dynamic Parallelism: introduction

- Introduced with CUDA 5.0
- makes it possible to launch kernels *from threads running on the device*:
  - threads can launch more threads
  - An application can launch a coarse-grained kernel which in turn launches finer-grained kernels to do work where needed
- avoids unwanted computations while capturing all interesting details

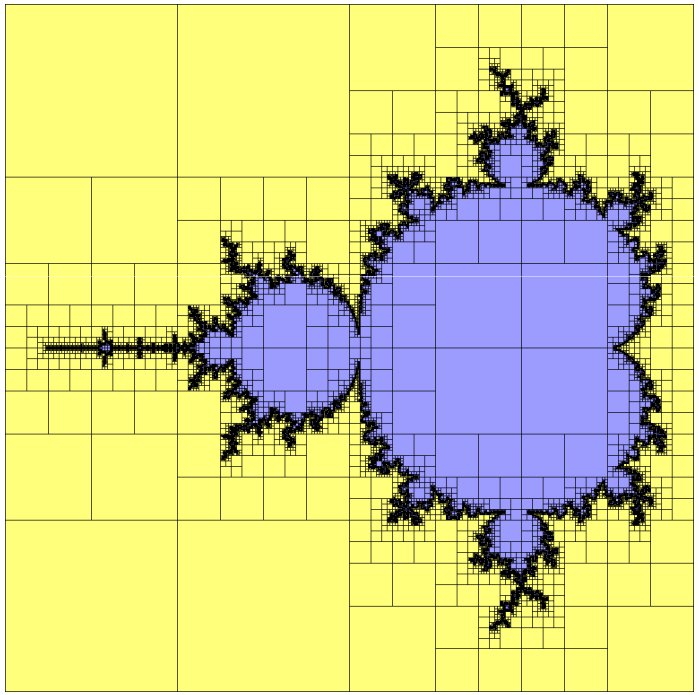


A fluid simulation that uses adaptive mesh refinement performs work only where needed

Where it is useful

- Dynamic parallelism is generally useful for problems where nested parallelism cannot be avoided
- This includes, but is not limited to, the following classes of algorithms:
  - algorithms using hierarchical data structures, such as adaptive grids;
  - algorithms using recursion, where each level of recursion has parallelism, such as quicksort;
  - algorithms where work is naturally split into independent batches, where each batch involves complex parallel processing but cannot fully use a single GPU
- Dynamic parallelism is available in CUDA 5.0 and later on devices of Compute Capability 3.5 or higher

Example: Mariani-Silver algorithm

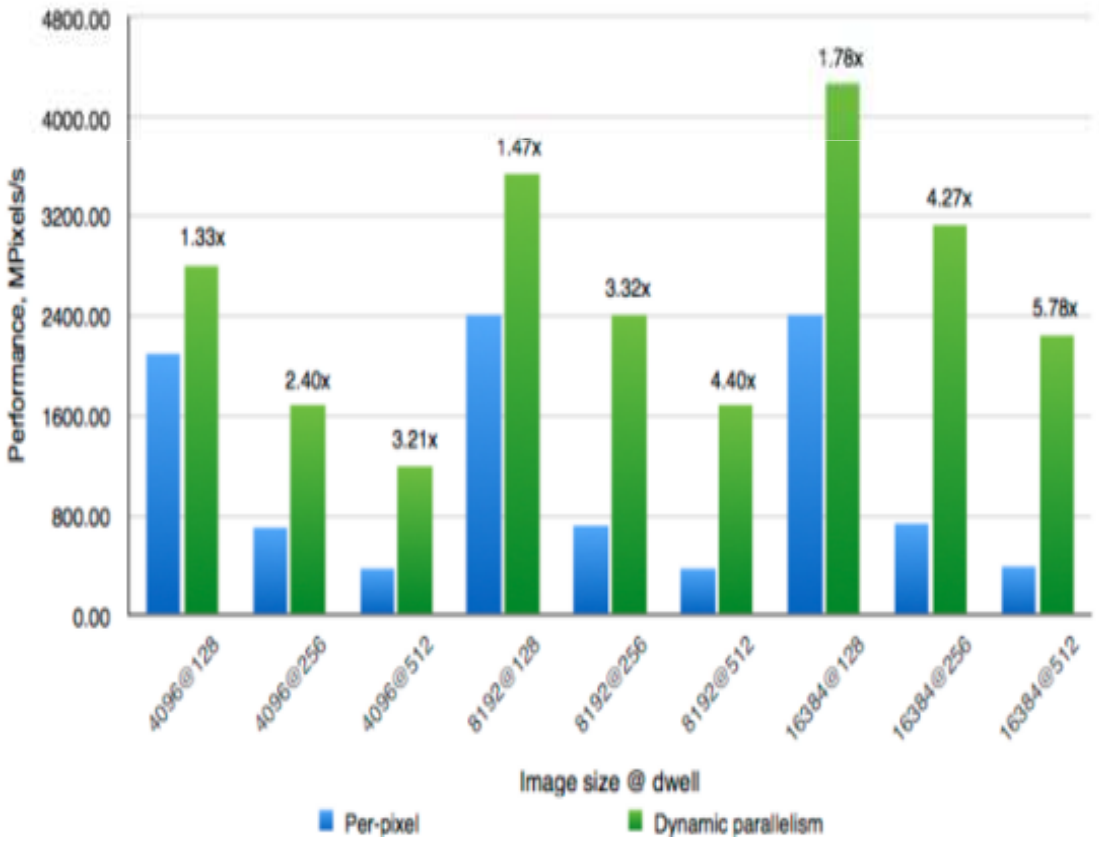


```
__global__ void mandelbrot_block_k(int *dwells,
                                   int w, int h,
                                   complex cmin, complex cmax,
                                   int x0, int y0,
                                   int d, int depth) {

  x0 += d * blockIdx.x, y0 += d * blockIdx.y;
  int common_dwell = border_dwell(w, h, cmin, cmax, x0, y0, d);
  if (threadIdx.x == 0 && threadIdx.y == 0) {
    if (common_dwell != DIFF_DWELL) {
      // uniform dwell, just fill
      dim3 bs(BSX, BSY), grid(divup(d, BSX), divup(d, BSY));
      dwell_fill<<<grid, bs>>>(dwells, w, x0, y0, d, comm_dwell);
    } else if (depth + 1 > MAX_DEPTH && d / SUBDIV < MIN_SIZE) {
      // subdivide recursively
      dim3 bs(blockDim.x, blockDim.y), grid(SUBDIV, SUBDIV);
      mandelbrot_block_k<<<grid, bs>>>
        (dwells, w, h, cmin, cmax, x0, y0, d / SUBDIV, depth+1);
    } else {
      // leaf, per-pixel kernel
      dim3 bs(BSX, BSY), grid(divup(d, BSX), divup(d, BSY));
      mandelbrot_pixel_k<<<grid, bs>>>
        (dwells, w, h, cmin, cmax, x0, y0, d);
    }
    cucheck_dev(cudaGetLastError());
```

Example: Mariani-Silver algorithm

- Performance comparison of a per-pixel Mandelbrot kernel and the recursive Mariani-Silver algorithm using Dynamic Parallelism
- the number above the bars indicates the speedup achieved by using dynamic parallelism

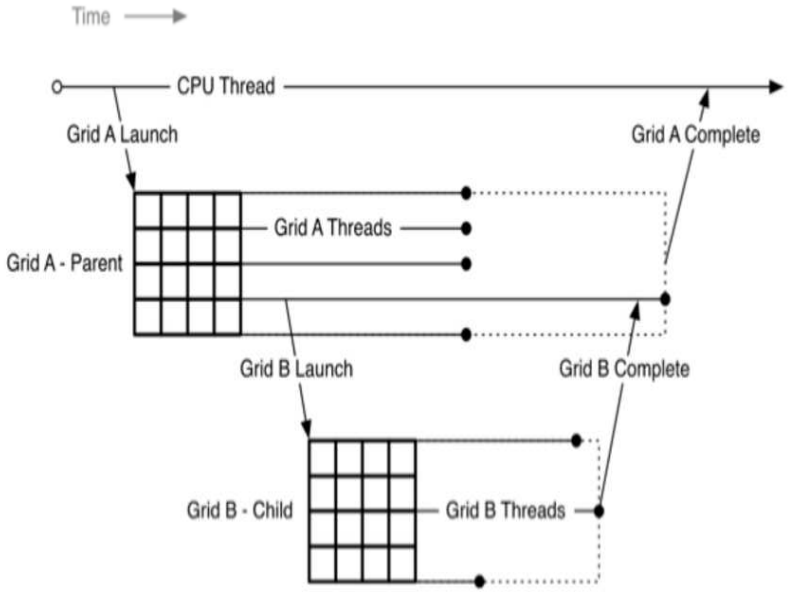


CUDA Dynamic Parallelism: Grid nesting

- A *parent grid* launches kernels called *child grids*
- A child grid inherits from the parent grid certain attributes and limits
  - L1 cache / shared memory configuration and stack size
- Every thread that encounters a kernel launch executes it

```
if(threadIdx.x == 0) {
  child_k <<< (n + bs - 1) / bs, bs >>> ();
}
```

- Grids launched with dynamic parallelism are *fully nested*:
  - child grids always complete before the parent grids that launched them, even if there is no explicit synchronization



Grid Synchronize

- Normally, parent kernel needs results computed by the child kernel to do its own work
- must ensure that the child grid has finished execution before continuing
- need to explicitly synchronize using **`cudaDeviceSynchronize(void)`**
  - waits for completion of all grids previously launched by the thread block from which it has been called
  - Because of nesting, it also ensures that any descendants of grids launched by the thread block have completed
  - **`cudaDeviceSynchronize()`** returns any error code that has occurred in any of those kernels

Grid Synchronize

- Note: when a thread calls **`cudaDeviceSynchronize()`**, it is not aware which of the kernel launch constructs *has been already executed* by other threads in the block
  - if a real block-level synchronization is desired, queueing of the child grids should be ensured by calling **`__syncthreads()`** before calling **`cudaDeviceSynchronize()`**
- Similarly, **`__syncthreads()`** should be called afterwards
  - so that other threads can only continue execution after the synchronization on the child grids has been performed

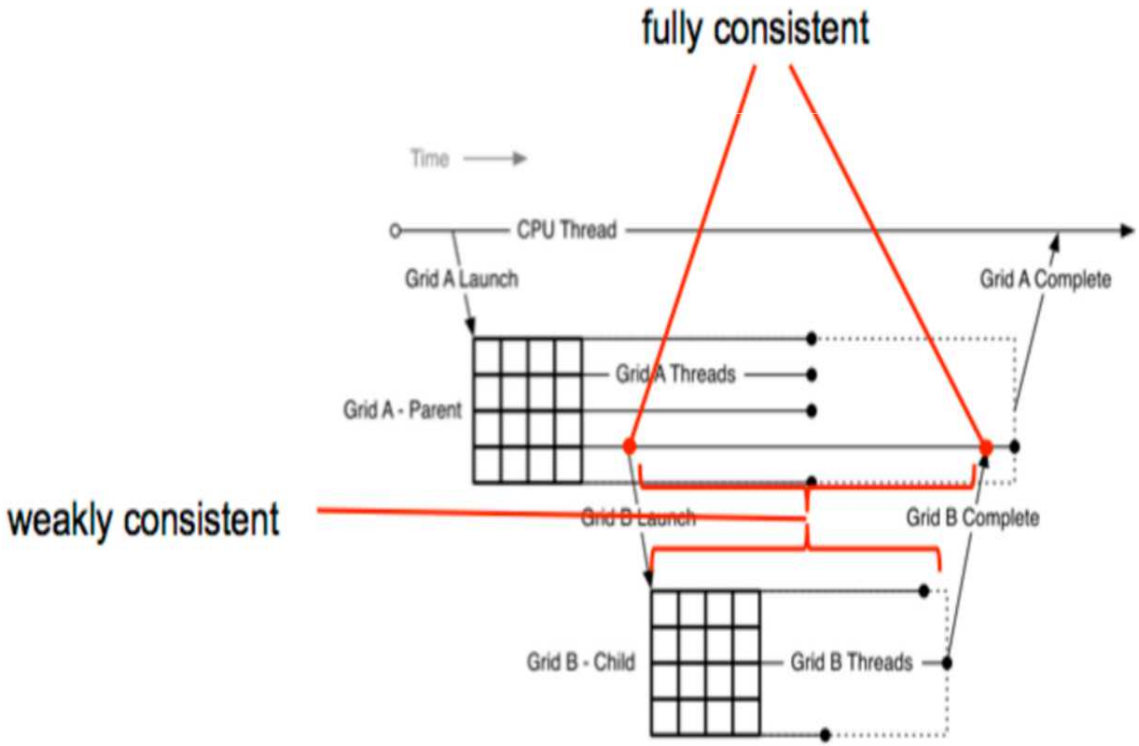
```
void threadBlockDeviceSynchronize(void) {
  __syncthreads();
  if(threadIdx.x == 0)
    cudaDeviceSynchronize();
  __syncthreads();
}
```

Grid Synchronize

- In general, calling **`cudaDeviceSynchronize()`** is expensive
  - it can cause the currently running block to be paused and swapped to device memory
  - call it only when necessary;
  - **`cudaDeviceSynchronize()`** should not be called at exit from a parent kernel, as implicit synchronization is performed anyway

Memory Consistency

- Guaranteed by CUDA Device Runtime:
  - parent and child grids have a ***fully consistent*** view of global memory (and zero-copy host memory) when the child starts and ends



Memory Consistency

- But, the view of global memory is *not consistent* when the kernel launch construct is executed
  - E.g. in the following code example, it is not defined whether the child kernel reads and prints the value 1 or 2

```
__device__ int v = 0;

__global__ void child_k(void) {
  printf("v = %d\n", v);
}

__global__ void parent_k(void) {
  v = 1;
  child_k <<< 1, 1 >>>> ();
  v = 2; // RACE CONDITION
  cudaDeviceSynchronize();
}
```

Passing pointers to child grids

- The table below summarizes the limitations on the classes of pointers that can be passed to child kernels

Can be passed	Cannot be passed
Global memory (including `__device__` variables and memory allocated with `malloc`)	Shared memory (`__shared__` variables)
Zero-copy host memory	Local memory (including stack variables)
Constant memory (inherited and not writable)	

Passing pointers to child grids

- The results of dereferencing a pointer in a child grid that cannot be legally passed to it are undefined
  - The following code on the left is illegal (it won't even compile), while the code on the right is legal
  - The compiler check could be bypassed somewhat, but because the pointer to local memory can reference a memory location that is legal in the child grid, this can lead to data corruption

```
// common __global__ void child_k(void *p) { // ... *p = res; // ... }
```

Wrong code

```
__global__ void parent_k(void) {
  // ...
  int v = 0;
  child_k <<< 1, 256 >>> (&v);
  // ...
}
```

```
__device__ int v;
__global__ void parent_k(void) {
    // ...
    child_k <<< 1, 256 >>> (&v);
    // ...
}
```

Correct code

Passing pointers to child grids

- Common pattern in CPU programs
  - not allowed on GPUs with dynamic parallelism
- Passing a pointer to a global variable is allowed
  - it is hardly useful, as there are most likely many child grids, and only one global variable
- We may:
  - use the device **`malloc()`** function: it works but it may be slow, as the current device allocator is not scalable;
  - using pre-allocated data structures, provided that it is known in advance how many child grids will be launched;
  - using a custom or third-party memory allocator, which might be a (more difficult) approach to follow when allocations cannot be done in advance

Device streams and events

- By default, grids launched within a thread block are executed sequentially:
  - the next grid starts executing only after the previous one has finished
  - This happens even if grids are launched by different threads within the block
- Often, however, more concurrency is desired;
  - as with host-side kernel launches, we can use CUDA **streams** to achieve this

Recursion depth and device limits

- Two concepts of *recursion depth*:
  - nesting depth, which is the deepest nesting level of recursive grid launches
    - kernels launched from the host having depth 0;
  - synchronization depth, which is the deepest nesting level at which **`cudaDeviceSynchronize()`** is called

Recursion depth and device limits

- Usually, synchronization depth is the nesting depth less 1
  - but if not every level synchronizes explicitly, it can be lower
  - In the example code each kernel is launched with its nesting level passed in the parameter **`depth`**
  - Although the nesting depth is 6, the maximum synchronization depth is only 3 , because that is the deepest level that synchronizes explicitly

```
__global__ void recursive_k(int depth) {
  // up to depth 6
  if(depth == 6)
    return;
  // launch 1 kernel (2 if depth == 3)
  if(threadIdx.x == 0) {
    recursive_k <<< 1, 1 >>> (depth + 1);
    if(depth == 3) {
      cudaDeviceSynchronize();
      recursive_k <<< 1, 1 >>> (depth + 1);
    }
  }
}

// launch from host
recursive_k <<< 1, 1 >>> (0);
```

Recursion depth and device limits

- The current maximum synchronization depth, and thus the amount of memory reserved, is indicated by the **`cudaLimitDevRuntimeSyncDepth`** device limit
  - if **`cudaDeviceSynchronize()`** is called deeper than current maximum synchronization depth, it returns an error, and no synchronization happens
  - By default, memory is reserved for two levels of synchronization
- Note: in CUDA runtime, **`cudaLimitDevRuntimeSyncDepth`** limit is actually the number of levels for which storage should be reserved, including kernels launched from host
  - Thus, it should be set to maximum synchronization depth plus 1
- There is also a hardware limit on maximum nesting depth, and thus synchronization depth;
  - On Compute Capability 3.5, the hardware limit on depth is 24

Recursion depth and device limits

- Another important limit is the number of *pending* child grids, or grids that can be either running, suspended, or in launch queue waiting to run
  - Pending launch buffer is the data structure used to maintain the launch queue as well as track currently running kernels
  - If a kernel launch is executed when the buffer is full, the behavior depends on the version of CUDA used
- By default, space is reserved for 2048 pending child grids
  - this can be extended by setting the appropriate device limit, as in the following code:

```
cudaDeviceSetLimit(cudaLimitDevRuntimePendingLaunchCount, 32768);
```

Recursion depth and device limits

- In case of buffer full:
  - With CUDA 5, the grid is simply discarded
  - With CUDA 6+, a variable-size virtualized pool has been added to fixed-size pool for the pending launch buffer
  - The runtime first tries to add the newly launched grid to the fixed-size pool, and if it is full, it uses the virtualized pool
  - While this means that grids are queued successfully, the costs of using the virtualized pool are higher than the fixed-size pool
- Always be aware of how many grids are pending:
- Easy to launch a very high number of kernels even with a very low recursion depth:
  - this will consume lots of memory, and may impact performance and even application correctness

Dynamic Parallelism vs. Host Streams

- Dynamic parallelism is better than host streams:
  - *Avoid extra PCI-e data transfers*. The launch configuration of each subsequent kernel depends on results of the previous kernel, which are stored in device memory
    - for the dynamic parallelism version, reading these results requires just a single memory access, while for host streams a data transfer from device to host is needed
  - *Achieve higher launch throughput* with dynamic parallelism, when compared to launching from host using multiple streams
  - *Reduce false dependencies between kernel launches*.

    All host streams are served by a single thread, which means that when waiting on a single stream, work cannot be queued into other streams, even if the data for launch configuration is available. For dynamic parallelism, there are multiple thread blocks queuing kernels executed asynchronously

Dynamic Parallelism: caveats

- Don’t start a child grid with just a few threads:
  - causes severe under-utilization of GPU parallelism, since only a few threads in a warp would be active, and there would not be enough warps
  - GPU execution would be dominated by kernel launch latencies
- Nested parallelism can be visualized in a form of a tree
  - but the tree for dynamic parallelism will look very different from more traditional CPU tree processing, as summarized below:

Characteristics	**Tree processing**	**Dynamic Parallelism**
Node	Thin (1 thread)	Thick (many threads)
Branch degree	Small (usually < 10)	Large (usually > 100)
depth	large	small