Sebastian von Alfthan
Sami Ilvonen
Fredrik Robertsén

Introduction to CUDA programming

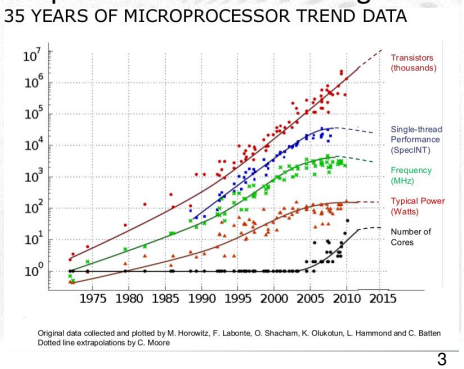
May 17-19, 2017
CSC – IT Center for Science Ltd, Finland

Introduction to CUDA programming

Introduction

Modern CPUs

- Very complicated, optimized for serial programs
  - Superscalar, out-of-order speculative execution with branch prediction logic and complicated cache hierarchies and vector units
- Single core performance is leveling off – performance is increasing through more parallelism
  - Multicore CPUs, in 2017 already up to 50 cores per node (Xeon), or even 70 (Xeon Phi)
  - Vector units are getting larger, AVX512 with 512 bit vectors
- Only a small part of the circuitry is doing the actual computation



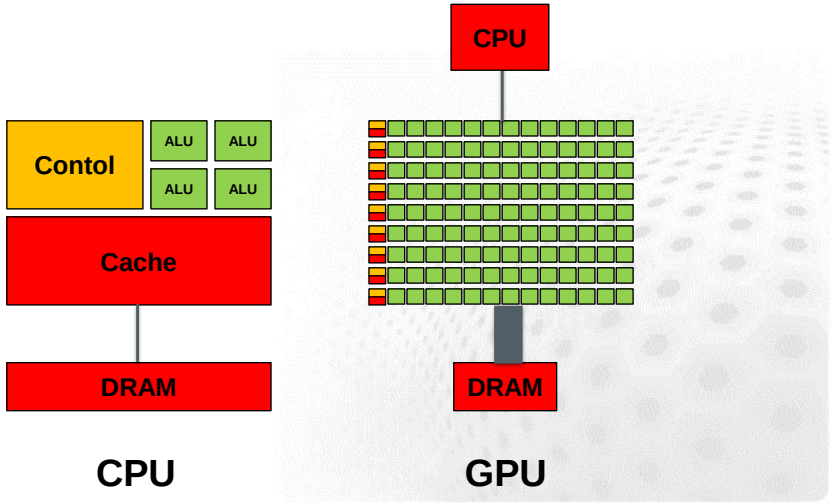
Accelerators

- Basic idea: move computationally intensive tasks to a specialized unit that is
  - Not suitable or slow for general purpose work load
  - Faster for the selected tasks
- Old example: floating point coprocessor 8087
- Common in consumer electronics
  - Audio/Video processing and compression
- Designing special hardware is very expensive

General Purpose GPU

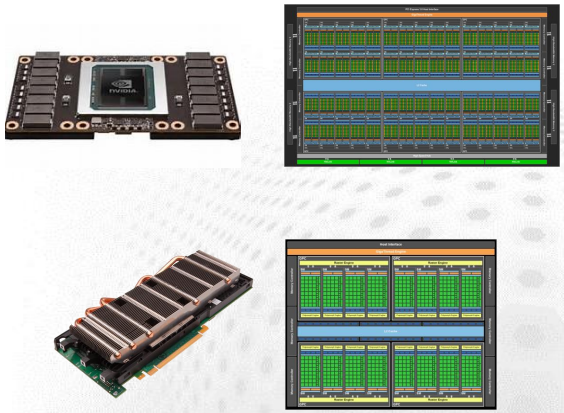
- GPU is Graphics Processing Unit that you can find inside a display adapter.
- GPUs have evolved rapidly in recent years, and now are also developed for highly parallel compute workloads
  - Much higher Flop/s than on CPU – more transistors for compute
  - Very high bandwidth to memory (700+ GB/s vs. 50 – 90 GB/s)
  - Higher flops/watts
- Compute workloads
  - HPC simulations
  - Deep learning currently the biggest market a
- Not a CPU – always needs a host CPU that it connects to with PCIe or NVLINK

Comparing CPU and GPU

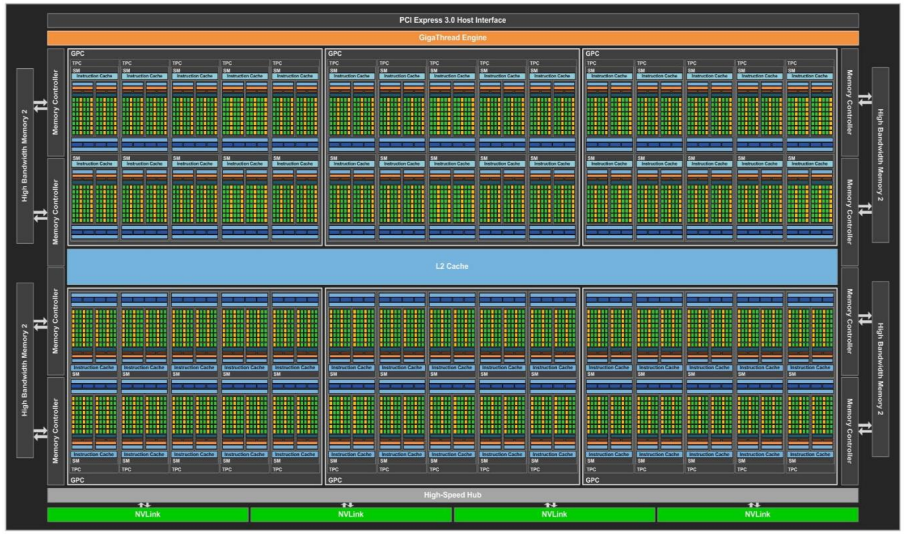


NVIDIA GPUs for HPC

- Pascal (e.g. P100)
  - 4.7 (DP)
    9.3 (SP)
    18.7 (HP) Tflop/s
  - 16 GB at 732 GB/s
- Kepler (e.g. K40)
  - 1.43 (DP)
    4.29 (SP) Tflop/s
  - 12 GB at 288 GB/s
- Fermi
- Tesla



Pascal Architecture



Pascal Architecture – streaming multiprocessor

Typical cluster in 2017

- Large amount of computing nodes
  - Distributed memory
  - Multicore processors (tens of cores per node)
- Fast network (interconnect)
  - Proprietary or Infiniband
- Programming model
  - Processes communicate via Message Passing (MPI)
  - Threads inside a node
- Accelerated cluster
  - 2 – 8 GPUS per node (typically 4)
  - Programming model: MPI + (threads) + CUDA

UTILIZING GPUS

Challenges

Applicability - Is your algorithm suitable for GPU?

Programmability - Is the programming effort acceptable?

Portability - Rapidly evolving ecosystem and incompatibilities between vendors.

Availability - Can you access a (large scale) system with GPUs?

Scalability - Can you scale the GPU software efficiently to several nodes?

How to Use GPUs

1. Use existing GPU software
2. Use numerical libraries with GPU support
3. Programming using directives
4. Native GPU code

Using Existing GPU Software

- NAMD, GROMACS, mumax3, Tensorflow …
- Pros
  - No implementation headaches for end users
- Cons
  - What if my science area/application is not supported?
  - Sometimes include only limited set of functionality
  - Is multi-node scalability good enough

Use Libraries with GPU Support

- CUBLAS, MAGMA, Thrust, CuFFT, ...
- Pros
  - Easy to implement in your programs
  - Algorithms in libraries are usually efficient
- Cons
  - Speedup limited by Amdahl's law and there is still transfer bottleneck

Directive Based GPU Code

- OpenACC
  - PGI compiler the preferred choice
  - May be the most prodctive choice at the moment
- OpenMP
  - Support since 4.0, improved support in 4.5
- Normal C/C++ or Fortran code with directives to guide compiler in creating GPU code
- Can also compile for CPUs, Knights Landing, …

Directive Based GPU Code

- Pros
  - Same code base as CPU version
  - Short time to solution
  - Portability is better due to different backends
- Cons
  - Generated code may not be as fast as hand-tuned CUDA

Native GPU Code

- CUDA, CUDA-Fortran (PGI), OpenCL
- Pros
  - Good control and best performance
- Cons
  - Requires most time
  - Portability (including performance)

CUDA vs OpenCL

- CUDA is NVIDIA specific
- OpenCL is standard
  - Writing OpenCL code that performs well on all platforms (NVIDIA, AMD, multicore CPUs) is difficult
- On NVIDIA hardware
  - CUDA is faster
  - According to NVIDIA
    - Remains NVIDIA's main programming interface
    - Will evolve faster than OpenCL

Summary

- Supercomputers with accelerators
- Efficient utilization of accelerators is challenging
- Programming methods for GPUs
  - CUDA, OpenCL
  - Directive-based approaches

Introduction to CUDA Programming I

Outline

- What is CUDA?
- Programming model
- Thread hierarchy
- Memory hierarchy
- Kernels
- Compiling
- Summary

WHAT IS CUDA?

CUDA

- **C**ompute **U**nified **D**evice **A**rchitecture
- **CUDA C** is a C/C++ language extension for GPU programming
  - PGI has developed similar Fortran 2003 extension
- **CUDA API** is the most up-to-date programming interface for NVIDIA GPUs
  - Other option: OpenCL (Khronos Group)
  - Two APIs: runtime and driver

Unified Device Architecture

- CUDA provides abstraction layer between different GPUs
  - Parallel Threads Execution (***ptx***) ISA and virtual machine
  - Ptx can be compiled to device binary code either at compile time or by the driver using JIT at runtime
- Important concept of ***compute capability***
  - Original G80 architecture supports CC 1.0
  - CC 3.5 supported by K40, and CC 3.7 by K80
    - Dynamic parallelism
  - CC 6.0 supported by P100 (Pascal)
    - Page faulting, half-precision
  - Many features only supported with most recent GPUs

CUDA C Code Consists of

Qualifiers
global, device, shared, local, constant, ...

Built-in variables
threadIdx, blockIdx, ...

Intrinsics
__syncthreads, __fmul_rn, ...

Runtime API calls
memory, device, execution management

Kernel launch

```
__device__ float array[128];
__global__ void kern(float *data) {
    __shared__ float buffer[32];

    ...
    buffer[threadIdx.x] = data[i];
    ...

    __syncthreads;
    ...
}

float *d_data;
cudaMalloc((void**)&d_data, bytes);

kern<<<1024, 128>>>(d_data);
```


CUDA APIs

- API: Application Programming Interface
  - Interface between the resource and computer program
- CUDA API includes functions for
  - Memory control (allocation, copying)
  - Synchronization and execution control
  - Hardware control and query
  - etc.

CUDA APIs

- CUDA Runtime API
  - User friendlier interface for application developers
  - Requires nvcc compiler
  - Runtime for host and device, device covered in dynamic parallelism lecture
- CUDA Driver API
  - Low-level interface for more detailed control
  - Much more complicated and verbose
  - Can be used with other C compilers
- We will only cover the runtime API!

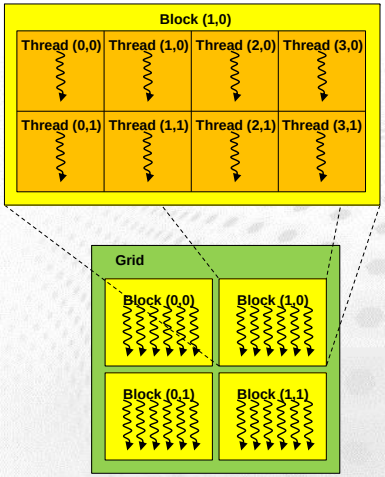
PROGRAMMING MODEL, THREADS

CUDA Programming Model

- GPU accelerator is called **device**, CPU is **host**
- Parallel code (**kernel**) is launched by the host and executed on the device by several threads
  - In modern CUDA kernel can also be launched from device, covered in the dynamic parallelism lecture
- Threads are grouped into thread blocks
- Program code is written from a single thread's point of view
  - Each thread has unique id
  - Each thread can diverge and execute a unique code path (can cause performance issues)

Thread Hierarchy

- Threads:
  - 3D IDs, unique in ***block***
- Blocks:
  - 3D* IDs, unique in ***grid***
- Dimensions are set at kernel launch
- Built-in variables for device code:
  - ***threadIdx***, ***blockIdx***
  - ***blockDim***, ***gridDim***



* Since CUDA4

Hardware Implementation, SIMT Architecture

- Maximum number of threads in a block depends on the compute capability (1024 on Kepler & Pascal)
- GPU multiprocessor creates, manages, schedules and executes threads in ***warps*** of 32
- Warp executes one common instruction at a time
  - Threads are allowed to branch, but each branch is executed serially
- Context switch is extremely fast, warp scheduler selects warps that are ready to execute → can hide latencies

Hardware Implementation (cont.)

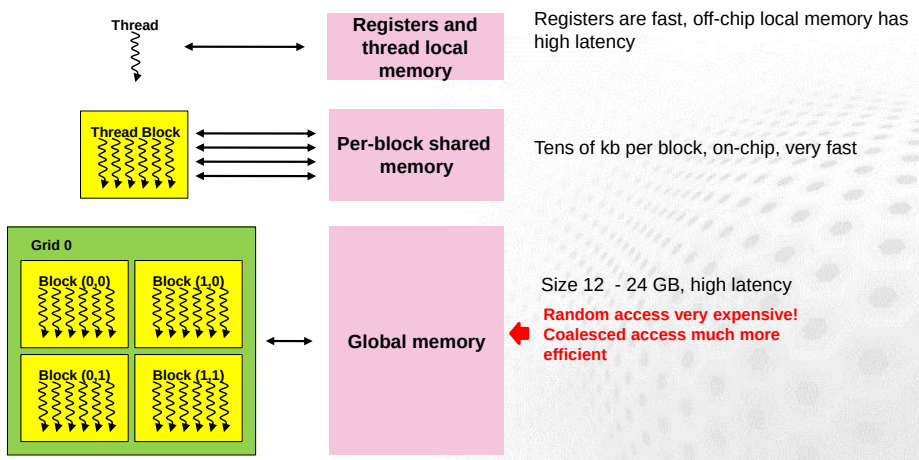
- The actual number of simultaneous threads that are executing is large, several thousands
- For maximum performance many more threads are needed
  - No context switching, can with low overhead juggle between threads
  - Thread scheduler can hide latencies efficiently only if there are enough threads waiting for execution - Many threads will be waiting for memory queries or register values
  - For example, Kepler and Pascal have 2048 slots for threads per multiprocessor, in total 30k+ threads on K40 and 110k+ on P100

PROGRAMMING MODEL, MEMORY

CPU and GPU Memories

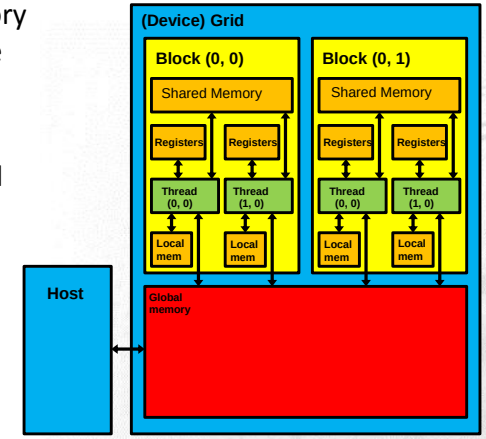
- Host and device have separate memories
- Host manages the GPU memory
- Usually one has to
  1. Copy (explicitly) data from host to the device
  2. Execute the GPU kernel
  3. Copy (explicitly) the results back to the host
- Data copies between host and device use the PCI bus with very limited bandwidth → minimize the transfers!

Device Memory Hierarchy



Memory Hierarchy

- Another view of the memory hierarchy of a CUDA device
- Arrows show the read and write permissions
- Host can only access global



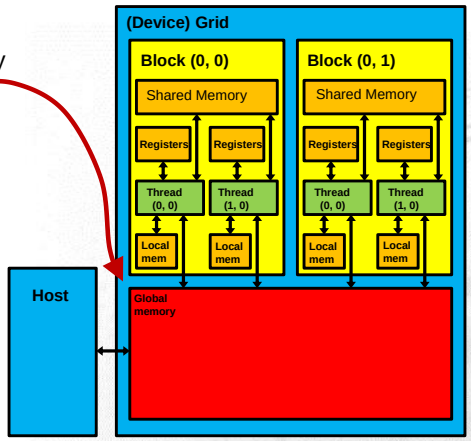
Memory Hierarchy

- Additionally there are more details in the memory hierarchy, some of which are architecture dependent
- Caches
  - L1 Cache on each streaming multiprocessor (64 KB on P100)
  - Shared L2 Cache (4096 KB on P100)
- Texture memory
- Constant memory

Allocating Device Memory

- **`cudaMalloc()`**
  - Allocate device global memory
- **`cudaFree()`**
  - Frees the allocated memory

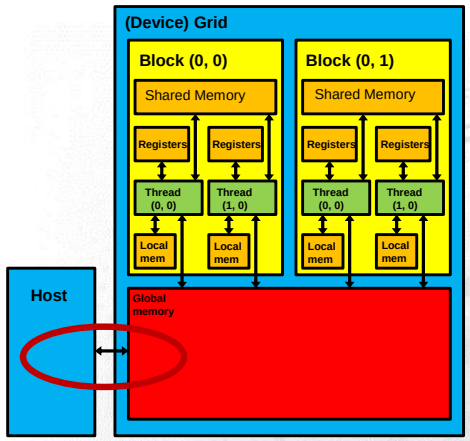
Note that the host code can not dereference device memory pointers!



Device-Host Data Transfer

- **`cudaMemcpy()`** transfers data from
  - Host to Host
  - Host to Device
  - Device to Host
  - Device to Device

This call blocks the execution of the host code. These is also an asynchronous copy function.



Example of Memory Operations

```
int main(void) {

   float *A = (float *) malloc(N*sizeof(float));
   float *d_A;
   cudaMalloc((void**)&d_A, N*sizeof(float));
   cudaMemcpy(d_A, A, N*sizeof(float), cudaMemcpyHostToDevice);
   ...
   float A0 = d_A[0];
   ...
   cudaMemcpy(A, d_A, N*sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(d_A);
   free(A);
   return 0;
}
```

Can not dereference device pointers in host code!

Memory Operations

cudaMalloc(void **devPtr, size_t size)

Allocate **size** bytes of device memory. Address returned in **devPtr**

cudaFree(void *devPtr)

Deallocate device memory

cudaMemcpy(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind)

Copies **count** bytes from **src** to **dst**. Kind is of the form cudaMemcpyDIRECTION, where DIRECTION is **Default**, HostToHost, HostToDevice, DeviceToHost or DeviceToDevice, and specifies the direction of the copy.

Default means that CUDA runtime uses unified virtual addressing (UVA) support to discover from the pointer value if the data resides on host or on a GPU

DEVICE CODE, KERNELS

Device Code

- C function with restrictions:
  - Can only dereference pointers to device memory
  - No static variables, no recursion
  - No variable number of arguments
- Functions must be declared with a qualifier
  - *__global__*: Kernel, typically called from CPU
    - Must return void
  - *__device__*: Called from *__device__* and *__global__* funcs
    - Can not be called from CPU
  - *__host__*: Can only be called by CPU
    - Can be combined with *__device__* qualifier

Calling GPU Kernel

- Special syntax:
  - kname<<<grid, block>>>(args)
  - kname is the name of the kernel function
  - grid determines the block hierarchy
  - block determines the thread hierarchy in a block
  - args is the list of arguments of the kernel
- grid and block can be either integers or struct (class) of type **dim3**
- There are two additional parameters, more about them later

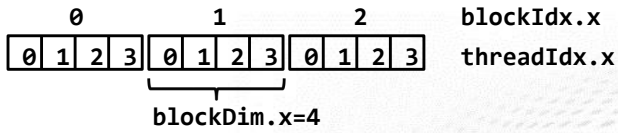
Kernel Call Example

3 blocks with 4 threads in each

```
__global__ void kern(int *A) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   A[idx] = idx;
}

void main() {
   // Allocate memories, copy values
   dim3 grid, block;
   block.x = 4;
   grid.x = 12/block.x;
   kern<<<grid, block>>>(d_A);
   // Copy results back
}
```

Result: A = {0,1,2,3,4,5,6,7,8,9,10,11}

Kernel Example (cont.)




```
__global__ void kern(int *A) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   A[idx] = blockIdx.x;
}
```

Result: A = {0,0,0,0,1,1,1,1,2,2,2,2}

```
__global__ void kern(int *A) {
   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   A[idx] = threadIdx.x;
}
```

Result: A = {0,1,2,3,0,1,2,3,0,1,2,3}

Compilation

- NVIDIA provides a compiler driver called *nvcc*
- *nvcc* separates the code for host and device
  - Host code is compiled with regular C/C++ compiler
  - Device code is compiled to ptx and/or cubin object
- *nvcc* or C/C++ can be used for linking
- Note that *nvcc* uses C++ front end to parse the program code

Compiling (cont.)

- Important options:

  -gencode arch=compute_XX,code=sm_XX

  determine the targeted architecture

  – Determine the target architecture for virtual architecture when generating PTX (arch), and further real architecture for which machine code is generated (code)

  – Add multiple gencode to support multiple architectures in the same binary

- For Taito K40 & K80 cards use

  – **nvcc -gencode arch=compute_35,code=sm_35 **
  -gencode arch=compute_37,code=sm_37

Summary

- CUDA programs consist of host and device code
- Device and host have separate memories
  – Memory allocations and transfers
- Device code is run parallel using threads organized into a grid of thread blocks
  – Kernel launch parameters
- Compilation

NVIDIA TOOLS FOR DEBUGGING AND PROFILING

Nvidia tool ecosystem

- Nvidia supplies a complete ecosystem for creating Cuda programs
  - Fewer things to fight with to get started compared to other GPU programming solutions
- Compiler, debugger, profiler, drivers, libraries, IDE
- Debugging with **cuda-memcheck**, **cuda-gdb** and **nsight**
- Profiling with **nvvp**, **nvprof**
- IDE with “all” the functionality built in **nsight**

PROFILING

Profiling

- Measure how well the GPU is performing
- Visualize what the GPU is doing during the program execution
- Primary tools:
  - Nvidia profiler, **nvprof**
  - Nvidia visual profiler, **nvvp**

- *COMPUTE_PROFILE* environment variable deprecated in CUDA 8

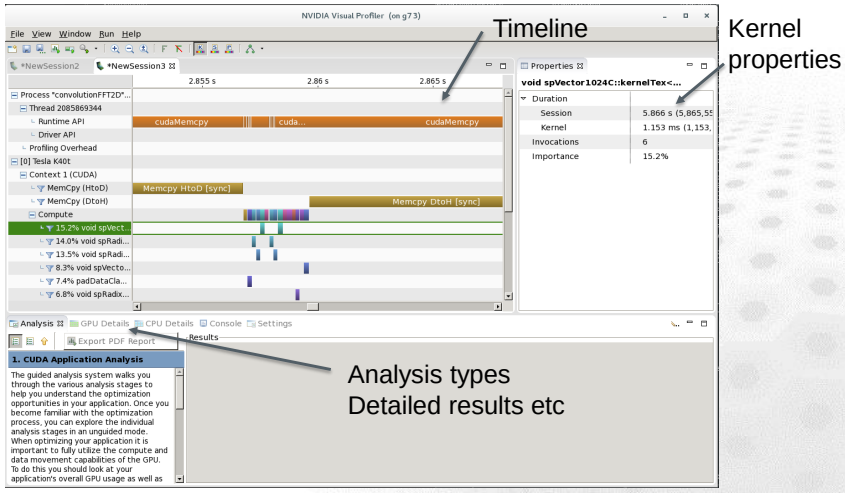
Prepare application

- API calls
  - nvtx library to mark ranges
- **-lineinfo** flag to mapping between ptx “assembler code” and source code
  - Useful for profiling specific parts of a kernel
- For more in-depth profiling, minimize your runtime

nvvp

- Nvidia visual profiler
- Available as both standalone application and part of the nsight IDE
- Useful for profiling (obviously) but also visualizing what your application is doing
- Really only good for one node/GPU programs
  - depending on how you do multi GPU
- Can be used to view other profilers output

Visual profiler timeline



Running nvvp

- On your local machine via the nvvp command
- On taito-gpu (for profiling)
  - Through nomachine/ssh with x forwarding
  - “module load cuda” (if not already loaded)
  - srun -n 1 -p gpu --gres=gpu:1 --x11=first nvvp
    - Remember to allocate a GPU and forward X

General workflow

- Select your application
- Generate timeline
  - Examine GPU usage to get more details for the whole run
    - Gives you occupancy and some additional info about the kernels
- Select a kernel to look closer at
  - Analyze bandwidth, compute, latency
  - Figure out what limits the performance

nvporf

- Command line program used for profiling CUDA programs
- Similar functionality to nvvp with no requirement for X
- Can be used to generate data for nvvp

Running nvprof

- Local machine
  - nvprof ./a.out
- Taito-gpu
  - Through ssh or nomachine
  - “srun -n 1 -N 1 --gres =gpu:1 nvprof ./a.out”
  - No need for X11, outputs everything to terminal

Workflow

- Run your application with *nvprof <application binary>*
- **--print-api-trace** will give you a timeline of all Cuda API calls
- **--print-gpu-trace** will print out what the GPU did
- **--metrics <metric names>** to collect specific metrics
  - **nvprof --query-metrics** to get list of possible metrics
    - Has do be run on the system with the actual GPU
- **--kernels** to limit the analysis to a specific kernel(s)

nvprof + nvvp

- nvprof quickly overload you with data or you want to profile a remote node, use nvprof to output data that can then be viewed in the visual profiler
- **-o <filename>** flag to nvprof
  - **%p** for process id
  - **%h** for the hostname
  - **--analysis-metrics** to generate everything needed for guided profiling in nvvp
    - Needs a separate run to generate the timeline
- Import into nvvp

Debugging

- cuda-gdb
  - Cuda capable version of GDB, will nor cover since GDB is difficult to teach in 1H
- cuda-memcheck
- Nsight IDE has debugging posibilites
- Build applications with **-g** flag
- **-lineinfo** flag to mapping between ptx "assembler code" and source code

cuda-memcheck

- Able to verify the correctness on the memory accesses of a CUDA program
  - Able to detect if accesses are outside arrays
  - Can detect potential race conditions in shared memory
  - Reports use of uninitialized data
  - Can detect synchronization issues in divergent code
- Command line tool, run your program through it
  - Parts of it can be used from cuda-gdb

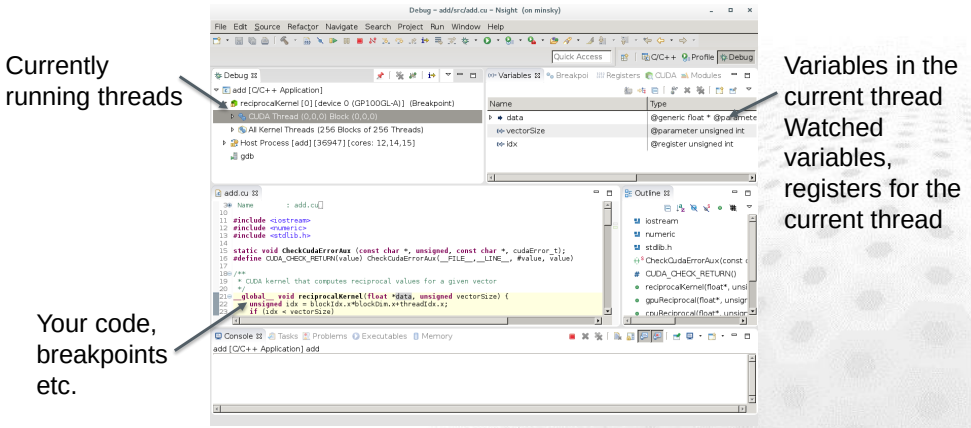
cuda-memcheck

- Run program with **cuda-memcheck --tool <tool used> <application>**
  - Where tool can be: memcheck, racecheck, synccheck or initcheck
- Will report which kernel and what thread in what block caused the issue
  - Will also tell you the address it tried to access for memcheck, initcheck and racecheck

nsight

- Full fledged Cuda/GPU integrated development environment
- Based on eclipse
- Available on windows, linux and mac
  - Windows also has it as a plugin to Visual Studio
- Has the debugging and profiling tools built in
- Can be used without an Nvidia GPU
  - Debugging and profiling requires a GPU
    - Can use remote node with GPU in it

Debugging with nsight



Use cases

- Helps you find errors in your code
- Logical errors
  - Step through code line by line
  - Monitor what values variables are assigned
  - Set breakpoints
- Memory errors
  - Enable cuda-memcheck
  - Will check your memory accesses

Summary

- Basics of profiling
  - Looking at what the application does
- Basic debugging
  - cuda-memcheck to check your memory accesses
  - nsight to be able to step through program and explore it while it is running

Introduction to CUDA Programming II

Outline

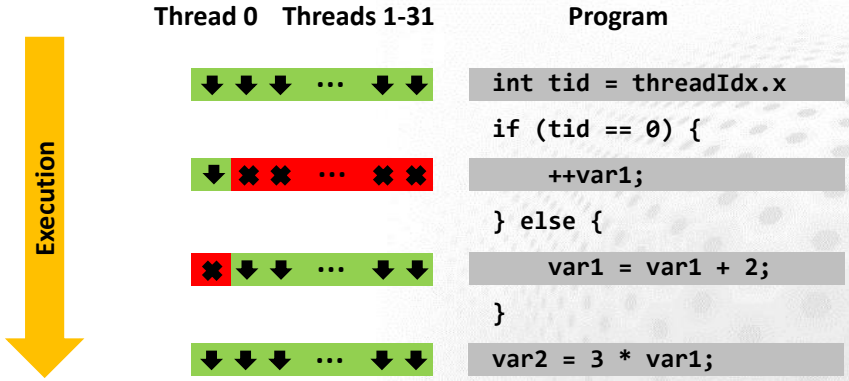
- Thread branching and synchronization
- Occupancy
- Coalesced memory access
- Error checking
- Summary

THREAD BRANCHING AND SYNCHRONIZATION

Thread branching

- Thread execution can branch according to e.g. thread index
  - All threads in the warp execute the *same* command
  - All threads do not have to participate
- Execution of different code paths is serialized
- Performance issues
  - Depends on the code paths and type of divergence, general suggestion is not to branch if possible

Thread branching cont.



Thread synchronization

- Blocks must be independent!
  - Can run in any order, concurrently or sequentially
- Different warps execute in arbitrary order
  - Weakly-ordered memory model, reads and writes by different threads can happen in different order
- Block level synchronization can be achieved with different synchronization intrinsics
  - `__syncthreads()`
  - `__threadfence()`, `__threadfence_block()`, `__threadfence_system()`

Thread synchronization cont.

- Synchronization between blocks is not well supported
  - Some level of coordination can be achieved using atomics
- In practice, the block level synchronization is done at kernel level
  - Returning to the host synchronizes the execution

OCCUPANCY

Occupancy

- Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps.
- Limited by your algorithm and resources, for example
  - Available registers
  - Amount of available shared memory

Occupancy (cont.)

- Good occupancy does not guarantee good performance!
- How ever, large number of active threads is needed for best performance
  - Thread execution model can hide latencies only if there are enough warps running

Block heuristics

- If the number of blocks > number of multiprocessors
  - All SMPs have a block to execute
- If the number of blocks / number of SMPs > 2
  - Concurrent execution of blocks in a SMP
  - Can hide **`__syncthreads()`**
  - Resource availability limits
- If the number of blocks > 100
  - Executed in “pipeline”

Threads

- Threads per block should be multiple of warp size
  - No under populated warps
  - Coalescing
- Should try to run as many warps as possible
  - Can hide latencies
- Heuristics
  - Minimum 64 (if still enough concurrent blocks)
  - More is better if resources do not limit
  - Experiment!

COALESCED MEMORY ACCESS

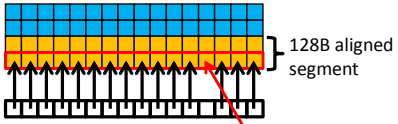
Coalesced memory access

- Global memory access has very high latency
- Threads are executed in warps, memory operations are grouped in a similar fashion
  - Memory access is optimized for coalesced access where threads read from / write to successive memory locations
  - Exact alignment rules and performance issues depend on the computing capability.
- Shared memory is better suited for more complicated data access patterns

Access patterns

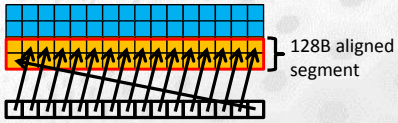
```
__global__ void offsetDemo(float *out, float *in, int oset)
{
    int tid = blocdIdx.x * blockDim.x + threadIdx.x + oset;
    odata[tid] = idata[tid];
}
```

oset = 0
Coalesced access



Note that access is coalesced also when all threads do not copy

oset = 1
Unaligned (but sequential) access



Coalesced access (cont.)

- The cost of non-coalesced access depends on the architecture
  - It used to be very high, recent cards are more forgiving
- Refer to the NVIDIA documentation for details

ERROR CHECKING

Error checking

- Almost all runtime functions return error code
- Asynchronous execution complicates error checking
  - Only errors that are trapped before the function return can be reported (e.g. parameter validation)
  - If something goes wrong, error will be returned by some subsequent, possibly unrelated function call
- For development phase it may be useful to add extra synchronization calls

Example of kernel error trapping

```
kernel_call<<<...>>>(...);
cudaDeviceSynchronize();
cudaError_t err = cudaGetLastError();
if (err != cudaSuccess) {
   printf("Error in kernel_call: %s\n",
       cudaGetErrorString(err));
   return 1;
}
```


Error checking (cont.)

```
cudaError_t cudaDeviceSynchronize(void)
```
Blocks until the device has completed all preceding tasks

```
const char* cudaGetErrorString(cudaError_t error)
```
Returns the message string from an error code

```
cudaError_t cudaGetLastError(void)
```
Returns the last error that has been produced by any runtime API call in the same host thread and resets it to cudaSuccess

```
cudaError_t cudaPeekAtLastError(void)
```
Returns the last error but does not reset it

Summary

- Synchronization, kernel level, independent blocks
- Warp divergence causes serialization
- Occupancy

Links to NVIDIA material

- NVIDIA CUDA main page
  https://developer.nvidia.com/cuda-zone
- CUDA documentation main page
  http://docs.nvidia.com/cuda/index.html

Introduction to CUDA Programming III

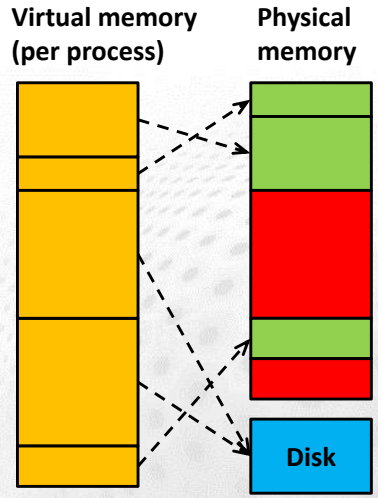
Outline

- Page-locked memory
- Streams
- Events
- Summary

PAGE-LOCKED MEMORY

Virtual memory system

- Modern operating systems utilize virtual memory
  - Memory is organized to memory pages
  - Memory pages can reside on swap area on the disk



Page-locked memory

- Normal **`malloc()`** call returns a pointer to virtual memory → swapping, page faults
- User can page-lock an allocated memory block to physical memory
  - Enables Direct Memory Access (DMA)
  - Higher transfer speeds between host and device
  - Copying can be interleaved with kernel execution
- Page locking too much memory can degrade system performance due to paging problems

Allocating page-locked memory

- Allocate directly using **`cudaHostAlloc()`** function instead of **`malloc()`**
  - Deallocate using **`cudaFreeHost()`**
- Page lock a range of memory allocated using normal **`malloc()`** call by using **`cudaHostRegister()`** function
  - Remove page-locking by calling **`cudaHostUnregister()`**
  - Remember page-alignment in allocation!

Asynchronous copies

- Normal **`cudaMemcpy()`** calls are blocking
  - They block the execution of host code until copying is ready
- In order to overlap copying and program execution one has to use asynchronous routines
  - Async suffix, for example **`cudaMemcpyAsync()`**
- User has to synchronize the program execution
- Requires page-locked memory!

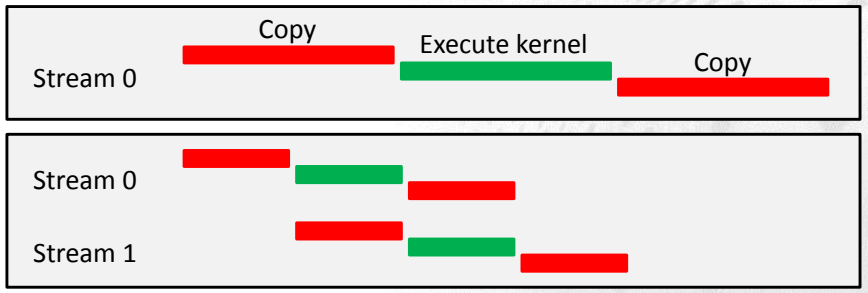
Concurrent execution

- Asynchronous by default:
  - Kernel launches
  - Device ↔ device memory copies
  - Host ↔ device memory copies of a block of 64 KB or less
  - Memory copies with suffix Async
  - Memory set function calls
- Devices can execute multiple kernels simultaneously

STREAMS

CUDA streams

- Stream is a sequence of operations that execute *in order* on GPU
- Can be used to synchronize e.g. overlapping computation and memory copies



Default stream

- When an explicit stream is not specified, kernel launches and host - device copies are issued to the *default stream*
  - Executed in order
- Before CUDA 7.0 the default stream was *NULL stream* that was common between all host threads
  - Different synchronization behavior
  - Blocked on all other streams, no concurrent execution of streams
- The default stream in CUDA 7.0 and up is a regular per host thread stream

Creating streams

- Stream is a variable of type **`cudaStream_t`**
  - It has to be initialized by **`cudaStreamCreate(&strm)`** before it can be used
- Remember to release the stream with **`cudaStreamDestroy(strm)`** when it is not used
- Host code can synchronize with the stream using **`cudaStreamSynchronize(strm)`**
  - Blocks until all CUDA calls associated to the given stream are completed

Specifying the stream for a kernel

- As mentioned before, kernel launches have four parameters
- The stream is the fourth parameter, so launching a kernel to a stream **strm** has following syntax

```
kern<<<grid, block, 0, strm>>>()
```


Streams example

```
cudaStream_t streams[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&streams[i]);
float* h_Data;
cudaMallocHost(&h_Data, 2*dsize);

for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(in_d_Data + i*dsize, h_Data + i*dsize, dsize,
        cudaMemcpyHostToDevice, streams[i]);
    MyKernel<<<100, 1024, 0, streams[i]>>>(out_d_Data + i*dsize,
        in_d_Data + i*dsize, dsize);
    cudaMemcpyAsync(h_Data + i*size, out_d_Data + i*dsize, dsize,
        cudaMemcpyDeviceToHost, streams[i]);
}

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


Streams

cudaError_t cudaStreamCreate(cudaStream_t *pstrm)

Creates a new stream, **pstrm** is the stream identifier.

cudaError_t cudaStreamDestroy(cudaStream_t pstrm)

Destroys and cleans up the stream specified by **pstrm**.

cudaError_t cudaStreamSynchronize(cudaStream_t pstrm)

Blocks until **pstrm** has completed all operations.

cudaError_t cudaStreamQuery(cudaStream_t pstrm)

Returns **cudaSuccess** if all operations in **pstrm** have completed, or **cudaErrorNotReady** if not.

Asynchronous memory functions

cudaError_t cudaMallocHost(void **ptr, size_t size, unsigned int flags)

Allocates **size** bytes of host memory that is page-locked.

cudaError_t cudaMemcpyAsync(void *dst, const void *src, size_t count, enum cudaMemcpyKind kind, cudaStream_t stream = 0)

Copies count bytes from addess **src** to **dst**. Argument **kind** is of the form **cudaMemcpyDIRECTION**, where **DIRECTION** is **HostToHost**, **HostToDevice**, **DeviceToHost** or **DeviceToDevice**, and specifies the direction of the copy. Call is asynchronous with respect to the host, so the call may return before the call is complete. Call works only with page-locked memory.

EVENTS

CUDA events

- Runtime API provides *events* to monitor device's progress and perform accurate timing
- Events are recorded asynchronously at any point of the program execution
  - An event has completed when all tasks or commands in a given stream have completed

Creating and using events

- Event is a variable of type **`cudaEvent_t`**
  - It has to be initialized by **`cudaEventCreate()`** before it can be used
- Remember to release the stream with **`cudaEventDestroy()`** when it is not used
- Event can be *recorded* using **`cudaEventRecord()`**
- Host code can be synchronized with stream using **`cudaEventSynchronize()`**
  - Blocks until the event is recorded on the device

Events

```
cudaError_t cudaEventCreate(cudaEvent_t *event)
```
Creates an event object **`event`**.

```
cudaError_t cudaEventDestroy(cudaEvent_t event)
```
Destroys an event object.

```
cudaError_t cudaEventRecord(cudaEvent_t event, cudaStream_t stream=0)
```
Records an event, default stream is stream 0.

```
cudaError_t cudaEventSynchronize(cudaEvent_t event)
```
Waits for an event to complete.

```
cudaError_t cudaEventElapsedTime(float *ms, cudaEvent_t start, cudaEvent_t stop)
```
Computes the elapsed time between two events (ms in milliseconds).

Events and timing

- Events have accurate timer information that can be used for performance monitoring
  - Resolution around 0.5 microseconds
- Time between two events can be queried with **`cudaEventElapsedTime(&time_diff,start,stop)`**
  - Result **`time_diff`** is in milliseconds
  - Both start and stop events have to be finished, otherwise an error is returned

Example

```
float time_diff_ms = 0.0;
cudaEvent_t start, stop;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel_call<<<...>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
cudaEventElapsedTime(&time_diff_ms, start, stop);
cudaEventDestroy(start);
cudaEventDestroy(stop);
```


Summary of CPU/GPU synchronization

- One can synchronize host thread and GPU by
  - **`cudaDeviceSynchronize()`**
    - Blocks until all previous CUDA calls in all streams of all host threads have completed
  - **`cudaStreamSynchronize(stream)`**
    - Blocks until all CUDA calls associated to the given stream are completed
  - **`cudaEventSynchronize(event)`**
    - Blocks until the event is recorded
  - **`cudaStreamWaitEvent(stream, event, flags)`**
    - Blocks the **`stream`** until **`event`** is completed

Summary

- Page-locked memory enables faster and asynchronous copies
- Streams and events can be used to synchronize programs execution, overlap copying and computing and timing
- Shared memory is better than global memory for non-coalesced memory access

KERNEL OPTIMIZATION

NOTE!!

- Some optimizations presented here might not be valid for future architectures
- Always consult the tuning guide (or similar) for the architecture you are working with
- Kepler: http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html
- Pascal: http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html

Before you spend hours optimizing the kernels…

- Data movement
  - Keep data on the GPU as long as possible
  - Data movement to and from the device is slow
- Occupancy
  - Use the occupancy calculator or nvvp to determine the optimal launch parameters
- Parallelism
  - Make sure you have sufficient parallelism in your program

Workflow

- Profile application, guided analysis is easier
  - Examine GPU usage, profiler should hint you where to look
  - Examine individual kernels, start with the “heaviest” one
    - Bandwidth
    - Compute
    - Latency
- Identify issue
- Fix the issue
- Sufficiently fast? Yes / no, redo the process

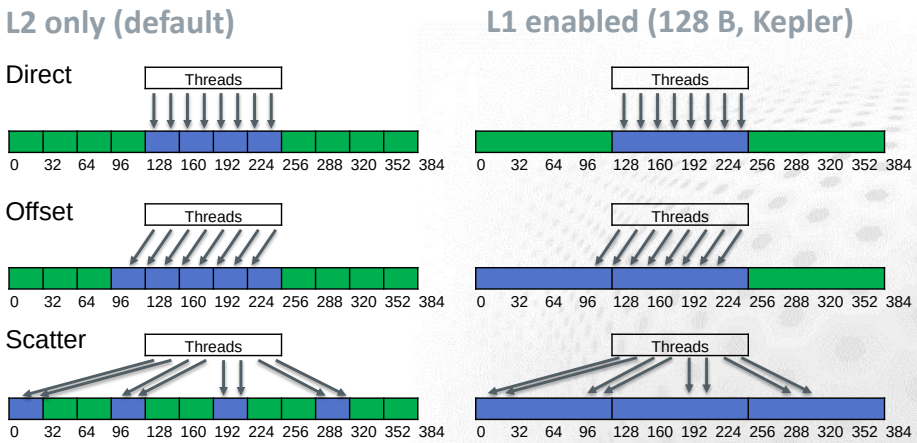
Global memory

- Important aspect of kernel performance
  - Using global memory wrong can be the cause of massive performance losses
- Metrics to look at
  - Global load and store efficiency
  - Device memory bandwidth utilization
- Nvvp can pinpoint the code lines causing the issue
  - Will tell you executed/ideal transactions for a given access

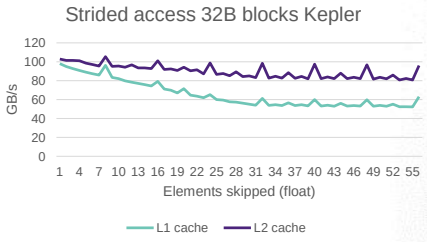
L1 cache

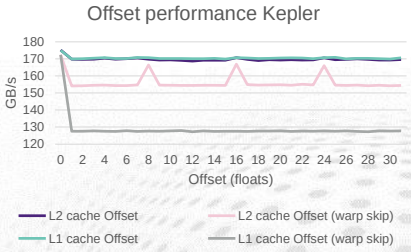
- Not used for global memory access by default on Kelper and newer
  - Used for register spills, per thread memory i.e. stack space
- Cached in L2, L2 load granularity 32 bytes
- Can be re-enabled with a compiler flag for entire program
  - For data reuse
  - Slightly higher bandwidth, fewer memory operations
  - Changes load granularity to 128 bytes
    - Pascal it stays at 32 bytes regardless of which how cached

Access patterns



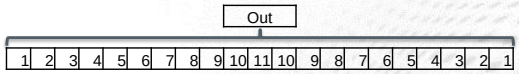
Global memory performance





Stencil example

- Simple 2D 21 point stencil
  - Accesses 20 neighboring points, scales them based on the stencil coefficients, sums everything into one value and stores it



- Significant data reuse, stencil coefficients and input data
- Using a single K40 GPU, Pascal tests on a P100 GPU
- Caching in L2 only: 1.02*10^9 updates/sec
- With L1 caching: 4.8*10^9 updates/s
- No real difference on Pascal, default caching works well

Texture memory

- A feature from the graphics world
- Separate cache on Kepler, part of L1 on Pascal
- Read only
- Special memory object based on a **cudaArray**
  - **cudaTextureObject_t, cudaCreateTextureObject()**
- Accessed with specific functions
  - **tex1D<>() , tex2D<>(), tex3D<>()**
- http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#texture-and-surface-memory

Texture memory functionality

- 2D textures can be used to optimize 2D access patterns
  - Stored in a beneficial way to enable 2D lookups
- Special indexing modes
  - Clamp/wrap around
  - Normalized coordinates
- Interpolation between values
  - Nearest point
  - Linear filtering

Read-only data cache

- Kepler (GK110) and up
- For data that is read-only for the lifetime of your kernel
- Uses the texture cache pipeline
  - Simpler to use than texture cache
  - Separate cache memory (Kepler), L1 (Pascal)
- Declare pointer with with **const* __restrict__** to hint at the compiler to use it
- Data loaded with **__ldg()** intrinsic (recommended)
- Shows up as texture reads in the profiler

Read only cache performance

- Back to the stencil case
  - Caching in just L2 performance: 1.02*10^9 updates/sec
  - Improve the performance without turning on L1
- Load stencil weights though read only cache
  - 2.97*10^9 updates/sec
- Load input data through read only cache
  - 1.18*10^9 updates/sec
- Load everything though read only cache
  - 3.54*10^9 updates/sec
  - 11% improvement on Pascal

Constant memory

- 64 KB memory on the device
- Useful for constant values all threads need
- **__constant__** qualifier
- **cudaMemcpyToSymbol()** to transfer data to constant memory
- Needs to be set before kernel launch
- All threads in warp should access the same address
  - Otherwise accesses will be serialized

Constant memory performance

- Revisit the stencil case
  - Stencil weights are the same for each thread and constant for at least the runtime of the kernel
  - Put them in constant memory
- Coefficients in constant memory
  - 3.69*10^9 updated/sec K40,
- And loading data through read only cache
  - 6.17*10^9 updated/sec
  - 66% improvement on pascal compared to initial version

Shared memory intro

- Separate on chip memory space in each multiprocessor
- Pre Pascal shared space with L1 cache
  - Can change size 16,32,48,64 KB
- All threads within a thread block have access to same memory space
- Used for
  - Thread to thread communication
  - Data reuse
  - “fixing” scattered memory access

Shared memory usage

- Allocated with __shared__ qualifier
  - Ex. **__shared__ int scratch[64];**
  - All threads share allocation
  - Size is for entire thread block
- __syncthreads() to synchronize data in shared memory
  - All running threads in block need to call __syncthreads()
    - Threads that have already exited are exempt
  - Only synchronizes threads in current block

Shared memory banks

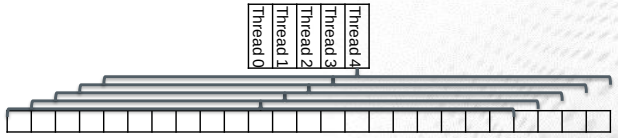
- Shared memory is divided into 32 banks
  - Idea being one for each thread in a warp
- 4 Byte banks by default
  - Changeable to 8 Bytes on Kepler (GK110/GK210)
- Threads accessing the same address in the same bank will be broadcast
- Threads accessing different addresses in the same bank will be serialized

Shared memory optimization

- Metrics
  - Shared efficiency
  - Shared bandwidth utilization
- Avoid bank conflicts
- **__synchthreads()** becomes expensive when used frequently
- Shared memory usage can affect occupancy
  - Tune launch parameters to fit current register and shared memory usage

Shared memory performance

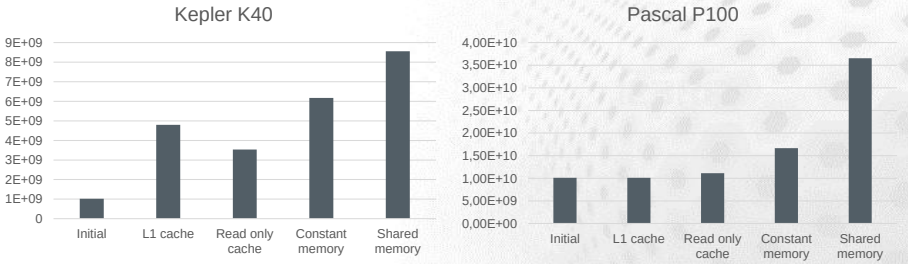
- The stencil case
  - Constant memory performance: 6.17*10^9 updated/sec
- Significant data reuse in the data we read in



- Storing the data in a shared buffer:
  - 8.56*10^9 updates/sec K40
  - 3.6x performance increase compared to initial on Pascal

Stencil performance summary

- Significant speedup from fairly small changes
  - Almost 10X on the Kepler system
  - 3.6X on the Pascal system, default caching on pascal works really well



Execution divergence

- All threads within a warp should perform the same instruction
  - If not branches are serialized
- Metrics
  - Warp execution efficiency
  - Compiled with **-lineinfo** will tell you which lines are issues

Floating point precision

- Kepler and consumer GPUs have more than 2x more single precision performance compared to double
  - Consumer GPUs are basically completely crippled when it comes to double precision
- Pascal half precision performance 2x of the single precision performance and 4x the double precision perf
- IF your code allows for it consider lower precision
  - ONLY if it actually does!!!

Intrinsic math functions

- In the case that speed is more important than precision use fast math functions
  - Really only for single precision
- Use __sinf() instead of sinf()
- -use_fast_math to change the whole program
- Information about precision
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#intrinsic-functions

Summary

- Global memory access patterns
- Specialized memory spaces
  - Constant memory
  - Read only cache
  - Texture memory
  - Shared memory
- Lower precision for better performance

Pascal and Unified Memory

Outline

- New features in Pascal
- Unified memory
  - Introduction
  - Usage
  - Prefetching
  - Hints

Pascal P100

- Pascal is the latest generation GPU architecture from Nvidia
- Many significant improvements
- NVLINK
  - 20 GB/s speed
  - Connects GPUs to each other in one node
  - Or even to CPU for Power based systems
- GPU can page-fault enabling useful unified memory usage

Pascal P100

- Much higher memory performance due to stacked HBM2 memory modules, 16 GB at 732 GB/s
- High floating point performance for double precision and support for half-precision (fp16)
  - 4.7 (DP)
  - 9.3 (SP)
  - 18.7 (HP) Tflop/s

UNIFIED MEMORY

Unified memory

- Unified memory
  - Introduced with Kepler and CUDA 6
  - Memory that one can access directly both in host and device code, allows CPU and GPU to USE same pointer (dereference)
  - Data is automatically transferred so that it is local to the computation on host or device

Unified memory vs. UVA

- Unified memory and Unified Virtual Addressing (UVA) are not the same thing
- UVA (since CUDA 4) provides a virtual memory address space
  - Allows cudaMemcpy, and other functions to know where the data is (host or device)
  - Does not allow host to dereference device pointer and vice versa
  - Enables “Zero-Copy” memory, where a device can access Pinned host memory over PCIe – no migration so this is slow!

Unified memory

- Why use it?
  - Makes it easier to port codes, especially codes with complex data structures
    - E.g. classes in C++, avoid deep copies
  - Data movement can be optimized once the code works
    - Through prefetches
    - Through Hints
    - Or by writing manual data movement code for performance critical data
  - Can make addressing larger-than-GPU arrays possible

Unified memory

- Drawbacks
  - Performance – data transfers less efficient that hand-tunes ones
  - Must still obey concurrency & coherency rules, not foolproof
  - Arguably only really useful on Pascal and later GPUs – limited in many ways on earlier cards

Allocating unified memory

cudaMallocManaged(void **devPtr, size_t size, unsigned int flags)

Allocate **size** bytes of managed memory. Address returned in **devPtr**

Default value for **flags** is **cudaMemAttachGlobal**: memory accessible on all devices & streams

cudaFree(void *devPtr)

Normal free function used to deallocate managed memory

Example – using unified memory on Kepler

```
const int maxLength=256;char *message;

cudaMallocManaged((void**)&message,
          sizeof(char) * maxLength);

 strcpy(message, "hello!");

 Kernel<<< 1, 1 >>>(message);


 cudaDeviceSynchronize();
 printf("On host: %s\n", message);

 cudaFree(message);
```

Allocate unified memory, pages populated on GPU

Set value on host, data migrates to CPU

Kernel launched, data migrates to GPU

Need to synchronize to touch data on host (Kepler)

Free unified memory

Example – using unified memory on Kepler

```
__global__
void Kernel(char *message) {
  printf("On device: %s\n", message);
  message[0] = 'd';
}


>nvcc test_um.cu -arch=sm_35  -o test_um
>./test_um
On device: hello!
On host: dello!
```

Unified memory on Kepler

- Can only allocate as much memory as on GPU, memory pages allocated before they are used
- Data migrates to GPU only on kernel launch – cannot migrate on-demand
- Strict coherency rules
  - Cannot touch any unified memory while any kernel is running
  - Must explicitly synchronize cudaDeviceSynchronize() , cuStreamSynchronize, cudaMemcpy…

Example – using unified memory on Kepler

```
const int maxLength=256;
char *message;
cudaMallocManaged((void**)&message, sizeof(char) * maxLength);
strcpy(message, "hello!");
Kernel<<< 1, 1 >>>(message);
printf("On host: %s\n", message);
cudaDeviceSynchronize();
cudaFree(message);
```

```
>nvcc test_um.cu -arch=sm_35  -o test_um
>./test_um
Bus error (core dumped)
```


Unified memory on Pascal

- 49 bit virtual addressing – can allocate more unified memory than physically on GPU(s)
- Page migration - on page fault pages migrate on-demand from anywhere on system (device or host)
- Pages populated and data migrated on first touch
- Concurrent access allowed – page-level coherency

Example – using unified memory on Pascal

```
const int maxLength=256;char *message;

cudaMallocManaged((void**)&message,
        sizeof(char) * maxLength);

strcpy(message, "hello!");

Kernel<<< 1, 1 >>>(message);

printf("On host 1: %s\n", message);
cudaDeviceSynchronize();
printf("On host 2: %s\n", message);

cudaFree(message);
```

Allocate unified memory, no pages anywhere

Set value on host, data allocated on CPU

Data migrates to GPU when it is accessed

concurrent access allowed, but be careful

Free unified memory

Example – using unified memory on Pascal

```
>nvcc test_um.cu -arch=sm_60  -o test_um
>/test_um
On host 1: hello!
On device: hello!
On host 2: dello!
```


Managed variable

- _ _managed_ _ qualifier
  - Declared that variable can be reference from both device and host
  - Has the lifetime of an application, global variable

Example – using _ _managed_ _

```
__managed__ int maxLength = 256;

__global__ void Kernel(char *message) {
  printf("On device: %s, maxLength %d.\n", message, maxLength);
  message[0] = 'd';
  maxLength=10;
}

int main(void) {
  …
  Kernel<<< 1, 1 >>>(message);
  cudaDeviceSynchronize();
  printf("On host: %s, maxLength set to %d\n", message, maxLength);
```


Example – using unified memory on Kepler

```
>./test
On device: hello!, maxLength 256.
On host: dello!, maxLength set to 10
```

Complex data structures

- If you have struct, and you want to transfer it to GPU you need to do a deep copy
- Allocate GPU structure
  - Allocate GPU storage for struct
  - Allocate storage for its members (and recursively of storage for members of members)
- Copy data to GPU
  - cudaMemcpy members
  - If member is pointer, both the GPU pointer value, and the actual data needs to be transferred

Example – deep copy

```
struct particles{
  double *xyz;
  int n;
};
particles h_p;  //host particle struct
h_p.n = 42;
h_p.xyz = (double*)malloc(h_p.n * 3 * sizeof(double));

particles *d_p; //device particle struct (pointer to)
double *d_xyz;  //device positions

// Allocate storage for device struct and the array inside
cudaMalloc(&d_p, sizeof(particles));
cudaMalloc(&d_xyz, h_p.n * 3 * sizeof(double));
```


Example – deep copy

```
//copy all members of struct, pointers are host pointers(!)
cudaMemcpy(&d_p, &h_p, sizeof(particles),
                cudaMemcpyHostToDevice);

//copy device pointers to device struct
cudaMemcpy(&(d_p->xyz), &d_xyz, sizeof(double*),
           cudaMemcpyHostToDevice);

//copy the data itself
cudaMemcpy(d_xyz, h_p.xyz, h_p.n * 3 * sizeof(double),
           cudaMemcpyHostToDevice);

//now kernels can be launched, etc…
```


Complex data structures with unified memory

- Unified memory can simplify life
  - Allocating the struct, and all of its dynamically allocated members in unified memory
  - This enables complex datastructures to automatically migrate

Example – deep copy with unified memory

```
struct particles{
  double *xyz;
  int n;
};
particles *p;  //particle struct pointer

//Allocate struct in unified memory
cudaMallocManaged(&p, sizeof(particles));
//and all of its members
p->n = 42;
cudaMallocManaged(&p->xyz, p->n * 3 * sizeof(double));

//now kernels can be launched, etc…
```


C++ classes with unified memory

- In C++ one can overload new and delete operator to use unified memory
- This enables you to write objects which migrate automatically, as long as all classes within a class do the same
- See https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/ for more details

Performance optimization aspects

- Finer grained concurrency with

```
cudaStreamAttachMemAsync(cudaStream_t stream, void *ptr,
                         size_t length=0, unsigned int flags=0);
```

- Attaches a unified memory region at ptr with a stream
  - CPU can access memory as long as all operations in stream have finished
  - Kernels in other streams are not allowed to touch data
  - Other streams can operate independently from this memory area

Performance optimization aspects

```
cudaMemPrefetchAsync(const void *ptr, size_t count, int
dstDevice, cudaStream_t stream);
```

- Prefetches data to the device where it will be used
- dstDevice can be GPU id or `cudaCpuDeviceId` for CPU
- If one does not prefetch, each new page that you access causes a page-fault, a costly operation (P100)

Performance optimization aspects: Pascal

```
cudaMemAdvise(const void *ptr, size_t count, enum
cudaMemoryAdvise advice, int device);
```

- When CPU and GPU needs to simultaneously access hints can be useful
- Advice can be
  - `cudaMemAdviseSetReadMostly`
  - `cudaMemAdviseSetPreferredLocation`
  - `cudaMemAdviseSetAccessedBy`
- See http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-tuning-usage for details

Summary

- Pascal brings fast memory, NVLINK and better unified memory
- Unified memory helps in porting real-world codes with complex data structures
  - Usage and allocation
  - Coherency on Kepler and Pascal
  - Deep copy
  - Performance optimizations
  - Not covered: Atomics & Multi-gpu considerations

References

- Parallel Forall blog
  - https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/
  - https://devblogs.nvidia.com/parallelforall/beyond-gpu-memory-limits-unified-memory-pascal/
  - https://github.com/parallel-forall/code-samples
- Presentations
  - https://www.olcf.ornl.gov/wp-content/uploads/2017/01/SummitDev_Unified-Memory.pdf
- Nvidia programming guide
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd

DYNAMIC PARALLELISM

Outline

- Introduction
- Calling kernels from device code
- Synchronization
- Memory management
- Limitations
- Summary

Dynamic parallelism

- What is dynamic parallelism?
  - Previously device code could be called from kernels, but kernels could only be launched from host code
  - Dynamic parallelism is an extension to the traditional CUDA programming model that enables a kernel to create and synchronize new work directly on the GPU
- Benefits:
  - Recursive algorithms are now allowed
  - Smaller synchronization cost

Example

Without dynamic parallelism

```
__global__ void k1(void) {
    …
}
__global__ void k2(void) {
    …
}

int main(void)
{
    k1<<<grid,blocks>>>();
    // has to return for check
    if (needed) {
        k2<<<1, 1>>>();
    }
}
```

With dynamic parallelism

```
__global__ void k1(void) {
    …
    // can call directly
    if (needed) {
        k2<<<1, 1>>>();
    }
}
__global__ void k2(void) {
    …
}

int main(void)
{
    k1<<<1, 1>>>();
}
```

Launching kernels

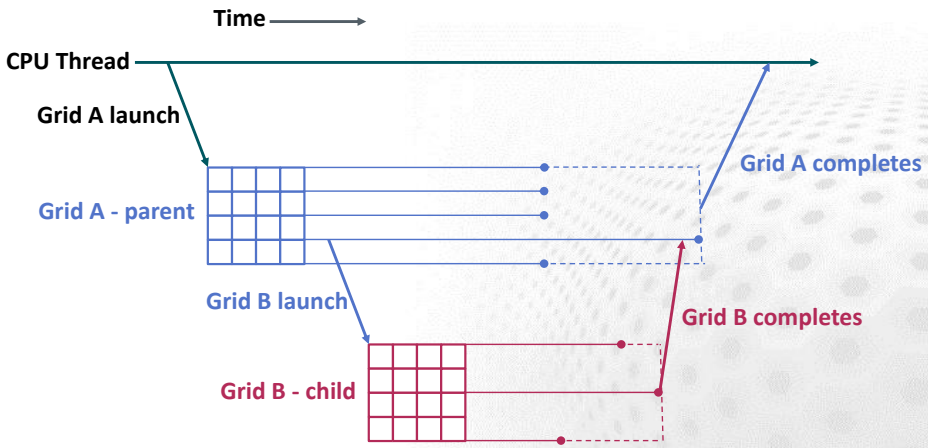
- Launches are per thread, not per block!
  - Each thread can launch new kernels
- Example: how many k2 kernels will be launched?

```
__global__ void k2(void) {
    printf("k2 launched\n");
}
__global__ void k1(void) {
    k2<<<1, 1>>>();
}

int main(void) {
    k1<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```


Synchronization

- Grids launched with dynamic parallelism are *fully nested*
  - Child grids always complete before parents
- If parent needs the results from children, it must ensure that the child has finished by calling **`cudaDeviceSynchronize()`**
  - Note that this call is also per thread!
- When block level synchronization is needed, a call to **`__syncthreads()`** is also needed
- No synchronization between blocks

Kernel launch nesting



Synchronization continued

```
__global__ void kern(void) {
    if (threadIdx.x == 0)
        child_kern<<<1, 1>>>();
```

`child_kern` results visible to (in same block):

← thread 0: No other threads: No

```
    // Sync with child_kern, but not with other threads
    cudaDeviceSynchronize();
```

← thread 0: Yes other threads: No

```
    // Sync with other threads in the block
    __syncthreads();
```

← thread 0: Yes other threads: Yes

```
    // Use the results computed by child_kern
    use_child_kern_results();
}
```

Compiling

- For example:

```
nvcc -arch=sm_35 -rdc=true -o demo demo.cu
```

- **`-rdc=true`**
  - Compiles *relocatable device code*, which is needed for dynamic parallelism
  - Can have a small negative impact on performance, do not use if not needed
- Note that CUDA 6.0 requires also device runtime linking flag **`-lcudadevrt`**

Memory model

- Global memory: visible to both parent and child
- Shared memory and local memory: private to child/parent, cannot be passed between each other
- Global memory is consistent between child and parent when
  - Child is called *(parent -> child)*
  - Child completes and **`cudaDeviceSynchronize()`** is called *(child -> parent)*

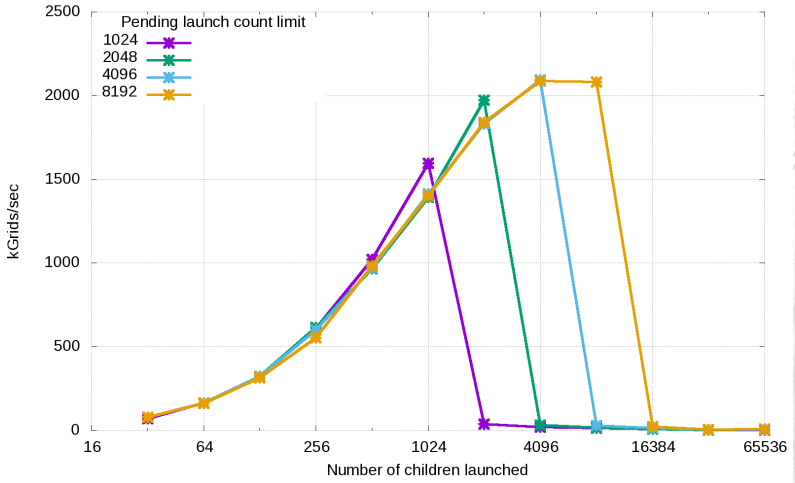
Recursion depth and device limits

- Dynamic parallelism has to concepts of recursion depth
  - **Nesting depth**: deepest nesting level of recursive kernel launches. Kernels launched by host have level 0
    - As of CC 3.5, the hardware limit is 24 levels of nesting
  - **Synchronization depth**: deepest nesting level at which **`cudaDeviceSynchronize()`** is called
    - Default maximum synchronization depth is two
    - Can be extended, but each new level can consume up to 150 MB of global memory

Pending kernel launches

- *Pending launch buffer* is a structure that is used to maintain information on the launch queue and currently running kernels
  - If kernel is launched when the buffer is full, the runtime will use *virtualized bool* (with CUDA > 6.0)
  - Cost of using virtualized bool is high
  - Know how many kernels you are going to launch!

Pending launch count limit



Using streams in device code

- By default, grids launched within a thread block are executed sequentially, even when grids are launched by separate threads
- More concurrency can be achieved using streams
- All device streams are non-blocking and they are per thread block only
- No per-stream synchronization on device streams
  - Only **`cudaDeviceSynchronize()`**
- Device and host streams can not be mixed

Error handling

- Device runtime supports error checking
- For *device-side exceptions* (dereferencing invalid address, etc.) error in a child grid will be returned to the host instead of the parent!

Allocating memory

- **`cudaMalloc()`** and **`cudaFree()`** are supported in device environment, but they have different semantics
- Can not mix host and device malloc and free

	`cudaMalloc()` on Host	**`cudaMalloc()`** on Device
`cudaFree()` on Host	Supported	Not supported
`cudaFree()` on Device	Not Supported	Supported
Allocation limit	Free device memory	*cudaLimitMallocHeapSize*

LIBRARIES

CUDA libraries

- CUDA toolkit includes a large number of libraries:
  - **cuFFT** Fast Fourier Transforms
  - **cuBLAS** Basic Linear Algebra Subroutines
  - **cuSPARSE** Sparse Matrix Routines
  - **cuSOLVER** Dense and Sparse Direct Solvers
  - **cuRAND** Random number Generation
  - **nvGRAPH** Graph Analytics Library
  - **Thrust** Templated Parallel Algorithms & Data Structures
  - **CUDA Math Library**
- Also several 3rd party toolkits and libraries

cuFFT

- 1D, 2D and 3D transforms of complex and real types
- Familiar API similar to FFTW
- Streamed asynchronous execution
- In-place and out-of-place transforms
- Thread-safe, can be called from multiple host threads

cuBLAS

- Complete support for all 152 standard BLAS routines
- Single, double, complex and double complex datatypes
- Supports CUDA streams and multiple GPUs
- Batched GEMM and LU
- Device API that can be called from CUDA kernels

cuRAND

- Host API for generating random numbers in bulk on the GPU and inline implementation that allows use inside GPU functions and kernels, or in your host code
- Four algorithms (both pseudo and quasi)
  - Multiple distributions

Thrust

- Thrust is a C++ template library for CUDA
  - Containers **`host_vector`** and **`device_vector`**
  - Algorithms, for example transform, fill, reduce, sort, copy_if, remove, remove_if, partition…
- Most recent versions support many C++11 features including lambda functions

Summary

- Dynamic parallelism enables kernel calls from kernels
- Can write recursive calls
- Easier to write adaptive algorithms
- Limitations
  - Synchronization and memory model
  - Levels of nesting and synchronization
  - Pending launch count
- CUDA toolset includes an extensive collection of accelerated libraries for common tasks

MULTI GPU PROGRAMMING

Outline

- Introduction
- Device management and context
- Programming models
- Device-to-device communication
- Summary

Introduction

- It is common to have several GPUs in single node (or workstation)
  - Share (and save) resources (disks, power units, e.g.)
- Advantages of Multi-GPU programming
  - More memory
  - More processing power
- Disadvantages
  - Programming is more complicated

DEVICE MANAGEMENT, CONTEXT

CPU-GPU context

- Context has to be established before GPU calls are done
- CUDA resources are allocated per context
- A context is established by CUDA runtime call
  - **cudaMalloc()**, **cudaFree()**, kernel launch...
- Context is destroyed when
  - **cudaDeviceReset()** is called
- CPU thread associated with the context is terminated

CPU-GPU context (cont.)

- By default, the model for runtime application is one context per device per process
  - Threads of the process share the context
  - Several processes can create contexts for single device
- The driver can limit the creation of contexts
  - Administrator can change the default mode
  - compute-exclusive-thread mode
  - compute-exclusive-process mode

Selecting device

- Driver associates a number for each CUDA-capable GPU starting from 0
- Device query and selection functions:
  - **cudaGetDeviceCount()**
  - **cudaSetDevice()**
  - **cudaGetDevice()**
  - **cudaDeviceReset()**
- **CUDA_VISIBLE_DEVICES** environment variable
  - Hides non listed GPUs

Device management

```
cudaError_t cudaGetDeviceCount(int *count)
    Returns in *count the number of CUDA capable devices.

cudaError_t cudaSetDevice(int device)
    Set device as the current device for the calling host thread.

cudaError_t cudaGetDevice(int *device)
    Returns in *device the current device for the calling host
thread.

cudaError_t cudaDeviceReset(void)
    Resets and explicitly destroys all resources associated with
the current device.
```

Querying device properties

- One can query the properties of different devices in the system using **cudaGetDeviceProperties** function.
  - No context needed
  - Can check e.g. amount of memory, compute capability, maximum dimensions for thread block, ...
  - Very useful for code portability
  - Some (workstation) systems may have display adapter in addition to computing GPU.

Device query

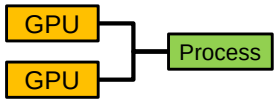
```
cudaError_t cudaGetDeviceProperties(struct cudaDeviceProp *prop,
int device)
    Returns in *prop the properties of CUDA capable device
device.

struct cudaDeviceProp {
        char name[256];
        size_t totalGlobalMem;
        size_t sharedMemPerBlock;
        int regsPerBlock;
        int warpSize;
        int computeMode;
        int concurrentKernels;
        int unifiedAddressing;
        ...
```


PROGRAMMING MODELS

Comparison of programming models (non exhaustive list)

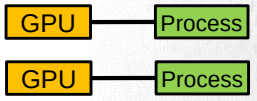
Single process for multiple GPUs

- Process switches the context and the active GPU
- Process synchronizes program execution



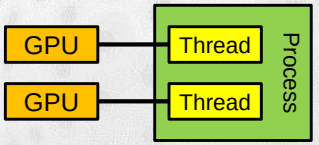
Single processes for single GPU

- Processes select their own GPUs
- Processes synchronize the program execution by messaging i.e. MPI



Single process - single thread for single GPU

- Each thread selects unique GPU
- Thread synchronization is used to sync the program execution



Single process for multiple GPUs

- Process can switch the active GPU using **cudaSetDevice()** function
- After setting the device, CUDA-calls are effective only on the selected GPU
  - Memory allocations and copies
  - Streams and events
  - Kernel calls
- One has to use asynchronous calls in order to occupy all devices

Single process for multiple GPUs (cont.)

- User has to keep track on different memory areas and streams, mixing variables between devices will fail in general
- Function **cudaStreamWaitEvent()** can be used for cross-device synchronization
  - Stream and event can be from different contexts

Single thread for single GPU

- Threads of a process can share the context
  - Access to same memory buffers, etc.
  - Usually the tasks of calling async copies and kernel execution are easily handled by single thread
- It can still be useful to have separate control thread for each GPU
  - How is your program synchronized and how the data is distributed and transferred?

DEVICE-TO-DEVICE COMMUNICATION

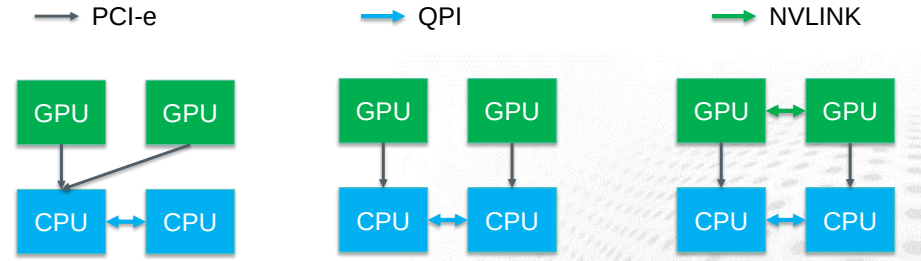
Device-to-device communication

- Devices have separate memories
- One can do the copy as GPU0-CPU-GPU1, but it is not efficient
- Other option: **cudaMemcpy()** function has an argument for type of copy
  - **cudaMemcpyDeviceToDevice** can be used to copy from one device to another
  - **cudaMemcpyDefault** with UVA

Peer to peer transfers

- Transfer data directly between GPUs without going through host memory
  - Lower latency, higher bandwidth
  - Requires GPUs to have PCI-e/NVLINK connection between them
- **cudaDeviceCanAccessPeer(int* canAccessPeer, int device, int peerDevice)**
- **cudaDeviceEnablePeerAccess(int peerDevice, unsigned int flags)**
- **cudaDeviceDisablePeerAccess(int peerDevice)**
- Falls back to normal copy through host if not available!

Peer to peer hardware requirements



Two or more GPUs connected to the same CPU
P2P transfers work
K80 nodes on Taito

Two GPUs connected to different CPUs
P2P transfers does not work
K40 nodes on Taito

Two or more GPUs connected with NVLINK
P2P transfers works
Some Pascal systems

Example

```
int peerAccess01;
int peerAccess10;

cudaDeviceCanAccessPeer(&peerAccess01, gpuId0, gpuId1);
cudaDeviceCanAccessPeer(&peerAccess10, gpuId1, gpuId0);

cudaSetDevice(gpuId0);
cudaDeviceEnablePeerAccess(gpuId1, 0);
cudaSetDevice(gpuId1);
cudaDeviceEnablePeerAccess(gpuId0, 0);

cudaMemcpy(buffGPU0, buffGPU1, size, cudaMemcpyDefault);
```

Peer to peer access

- Allows us to access memory from another GPU
  - Can pass pointer to data on GPU 1 to a kernel running on GPU 0
  - Will read data from other GPU over Pci-e bus
- Same requirement and setup as peer to peer transfers
- Really only useful for data used once
  - If you use the data on the other GPU more than once transfer it over

Example

```
cudaSetDevice(gpuID0);
cudaDeviceEnablePeerAccess(gpuID1, 0);
cudaSetDevice(gpuID1);
cudaDeviceEnablePeerAccess(gpuID0, 0);

__global__ void copyKernel(float *src, float *dst){
       const int idx = blockIdx.x * blockDim.x + threadIdx.x;
       dst[idx] = src[idx];
}

cudaSetDevice(gpuID0);
copyKernel<<<blocks, threads>>> (gpu0Buf, gpu1Buf);
cudaSetDevice(gpuID1);
copyKernel<<<blocks, threads>>> (gpu1Buf, gpu0Buf);
```


Summary

- Concept of context
- Device management
- Programming models
- Copying data between devices, P2P

Parallel Programming Using CUDA and MPI

Outline

- Very short MPI introduction
- Introduction to CUDA and MPI
- Compiling and linking
- Summary

INTRODUCTION

Message Passing Model

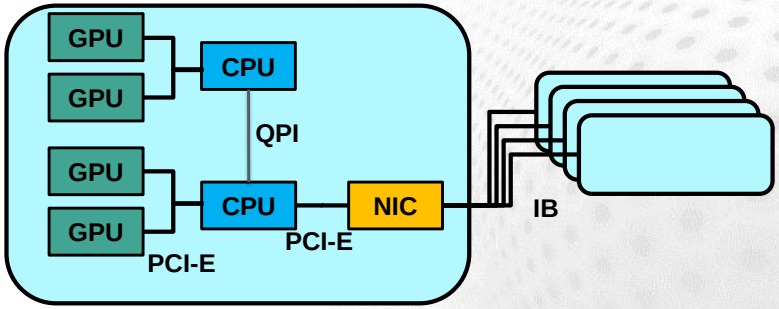
- Parallel program is launched as a set of independent, identical *processes*
  - Same program code and instructions
  - Processes can reside in different nodes or even different computers
- All variables and data structures are local to the process
- Processes can exchange data by sending and receiving *messages*

Message Passing Interface

- MPI is an API for communication between separate processes
  - The most widely used approach for distributed memory parallel computing
- MPI programs are portable and scalable
- MPI is flexible and comprehensive
  - Large, over 120 procedures
- Standardization by MPI-Forum.org

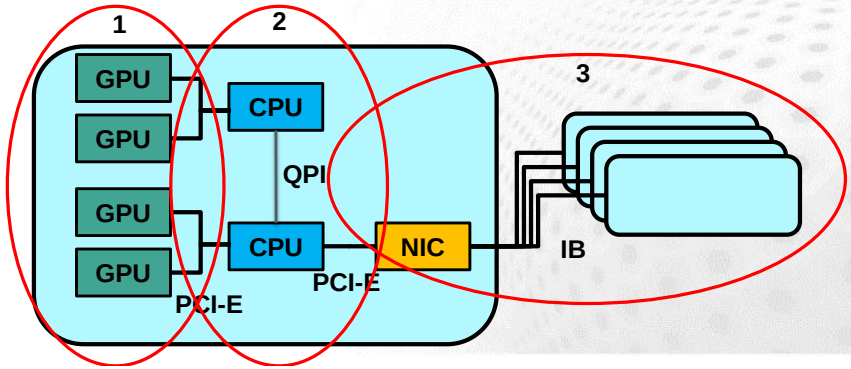
Introduction to MPI and CUDA

- Three levels of hardware parallelism
  1. GPU - threads on the multiprocessor
  2. Node - binding together GPU, CPU and interconnect
  3. Machine - Several nodes connected with interconn.



Introduction to MPI and CUDA

- Parallelization strategies
  1. CUDA for GPUs
  2. Threads (OpenMP) or MPI for CPUs
  3. MPI between nodes



MPI AND CUDA

MPI and CUDA

- CUDA and MPI can be considered separate
  - CUDA is used for GPU
  - MPI is used for parallelization over nodes
- Transferring data between devices – CUDA-aware MPI
  - Recommended to use CUDA-aware MPI library
  - Can directly use device pointer in MPI call
- Transferring manually
  - Sender: copy data from device to buffer
  - Sender: send host buffer data
  - Receiver: receive data to host buffer
  - Receiver: copy buffer data to the device

MPI and CUDA Difficulties: Scalability

- More challenging than traditional CPU machine
  - Higher computational power per node
  - Higher latency for GPU to GPU copies between nodes
- Efficient communication is non-trivial
  - Hiding latencies through overlapping computation and communication
  - Pipelined sends and receives when transferring manually
- Efficiency of CUDA-aware MPI varies

MPI and CUDA Strategies

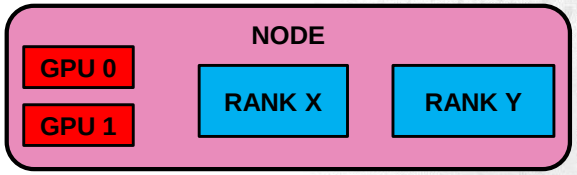
1. One MPI process per node,
2. One MPI process per GPU
3. Many MPI processes per GPU, only one uses it
4. Many MPI processes sharing a GPU

- Here we will discuss strategy 2 – one MPI process per GPU
- Can be combined with OpenMP
  - Utilize all cores on CPU for CPU computation
  - All GPU orchestration done by one (master) thread

ASSIGNING GPUS

Selecting the GPU

- Problem description:
  - If a node has more than one GPU, all processes in the node can access all GPUs of the node
  - MPI processes do not have a priori information on the other ranks in the same node
  - Which GPU should the MPI process select?



Selecting the GPU

- Simple approach
  - Use batch system options to assign the ranks in linear fashion to the cores
    - Node 1 ranks: 0 1 2 3
    - Node 2 ranks: 4 5 6 7
  - You may also assume that you know the number of processes per node to compute the rank on the node
    - **`int nodeRank = rank % processesPerNode`**

Selecting the GPU (cont.)

- Simple approach is not very robust - MPI-3 approach better

```
int gpuCount, nodeSize, nodeRank;
MPI_Comm nodeComm;

MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED, 0,
                    MPI_INFO_NULL, &nodeComm);
MPI_Comm_size(nodeComm, &nodeSize);
MPI_Comm_rank(nodeComm, &nodeRank);
cudaGetDeviceCount(&gpuCount);
if(nodeSize != gpuCount) {}//Handle error

cudaSetDevice(nodeRank);
```


MPI COMMUNICATION

MPI communication

- CUDA-aware MPI
  - Simply use device and/or host pointers as buffers in MPI calls
  - Uses UVA to see where data resides
  - Transfers in the background data along its most efficient path

```
if (rank == 0) {
    MPI_Send(dBuffer, size, MPI_BYTE, 1, 100, MPI_COMM_WORLD);
} else if (rank == 1) {
    MPI_Recv(dBuffer, size, MPI_BYTE, 0, 100, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
}
```


CUDA-Aware MPI Libraries

- UVA and GPUDirect enable the direct copy from GPU memory to the interconnect
  - No need for user specified buffers and explicit copies
  - Can use pipelining internally without end-user programming effort
  - Simplifies programming considerably
- Requires modifications to the internal implementation of MPI library for optimal performance

CUDA-aware MPI communication

- Benefits
  - By far easiest and clearest choice
  - With a correctly installed and setup system it gives best performance
- Cons
  - One may be able to get a bit more performance with manual tuning, at the expense of complexity
  - Not portable to a system without CUDA-aware MPI, but this is rare

COMPILING PROGRAMS WITH CUDA AND MPI

Compiling

- Both MPI and CUDA provide compiler wrappers
  - MPI compiler can not compile CUDA code
  - nvcc can not (by default) compile MPI programs
- Simple solution, separate MPI and CUDA
  - Compile to objects using e.g. mpicxx and nvcc
  - Link using mpicxx or nvcc
  - In practice, it is easier to add CUDA runtime library -lcudart to mpicxx linking than try to link MPI libraries with nvcc

Compiling (cont.)

- If code is separated, complicated copy-send operations become cumbersome
  - Can not call **`cudaMemcpy()`** before **`MPI_Send()`** in same routine
- It is possible to add most CUDA runtime calls (but no kernel execution) to MPI routines
  - Have to include the function prototypes and datatype definitions from **`#include <cuda_runtime_api.h>`**
  - Location of the header is system and installation dependent!

Summary

- CUDA-aware MPI makes MPI parallel GPU applications “easy”
  - Use same pointer
  - Performance good on well configured system
- Compilation