
[Morgan McGuire]
GPU Programming 101

Parallel Kernel

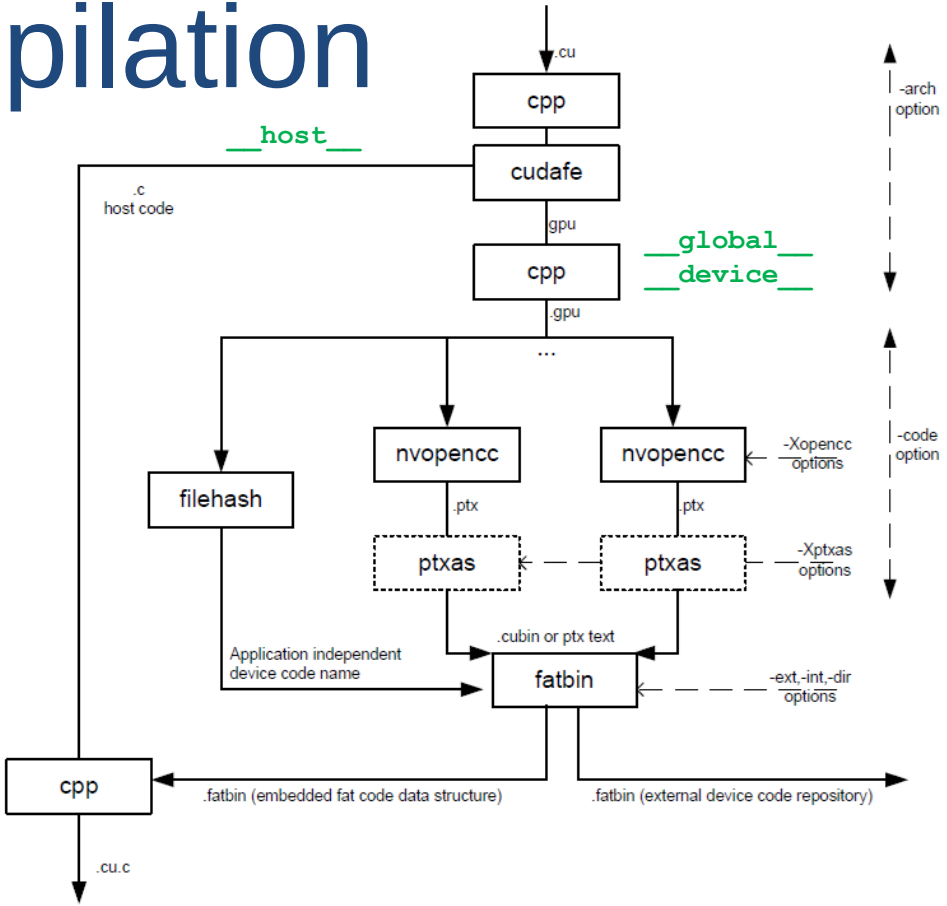
- Kernel is split up in blocks of threads

Kernel launch

```
__global__ void myfunction(float *input, float* output)
{
  uint id = threadIdx.x + blockIdx.x * blockDim.x;
  output[id] = input[id];
}

...
dim3 blockSize(128,1,1);
dim3 gridSize(12,1,1)
myfunction<<<dimGrid,dimBlock>>>(input, output);
```

Compilation



SCAN AND SORT

Parallel Prefix Sum - Scan

- **Input:** $x_0, x_1, x_2, x_3, \dots, x_n$
- **Output:** $y_0, y_1, y_2, y_3, \dots, y_n$

$$y_0 = x_0$$
$$y_1 = x_0 \oplus x_1$$
$$y_2 = x_0 \oplus x_1 \oplus x_2$$

associative, binary operator (e.g., +)

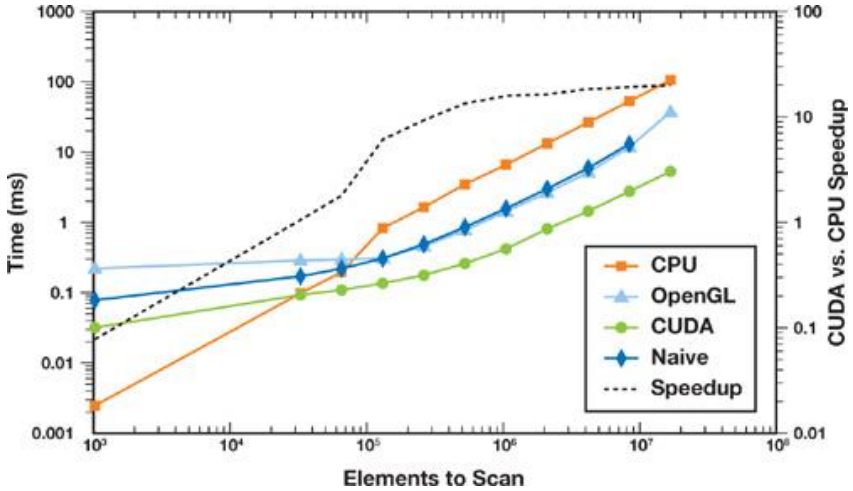
- Usecases:

  Compare strings, add multi precision numbers, evaluate polynomials, solve recurrences, **implement radix sort**, implement quicksort, solve tridiagonal linear systems, delete elements from an array, **dynamically allocate array elements**, perform lexical analysis, search for regular expressions, implement tree operations, label components in two dimensional images

`[2,3,3,1,0,1]` → `[2,5,8,9,9,10]`

Parallel Prefix Sum – Scan cont'd

- Can be efficiently implemented for parallel architectures [*Harris et.al. 2004]*



@ Geforce GTX 8800

Scan: Memory offset

Every thread produces a different number of elements
->Where to store them to fill up an array?

```
__global__ void analyse(uint *numel){
  ...
  numel[linid] = numelements();
}
__global__ void write(uint *numel, T* data){
  uint mynum = numelements();
  for(uint i = 0; i < mynum; ++i)
    data[numel[linid] + i] = genData(i);
}

analyse<<<x,y>>>(numel);
uint sum = scanExlusive(numel);
data = gpuMalloc<T>(sum);
write<<<x,y>>>(numel, data);
```

numel: [2,3,3,1,0,1]
10 = [0,2,5,8,9,9]

data:
[$d_{0,0}$,$d_{0,1}$,$d_{1,0}$,$d_{1,1}$,$d_{1,2}$,$d_{2,0}$,…]

Scan: threads for next kernel

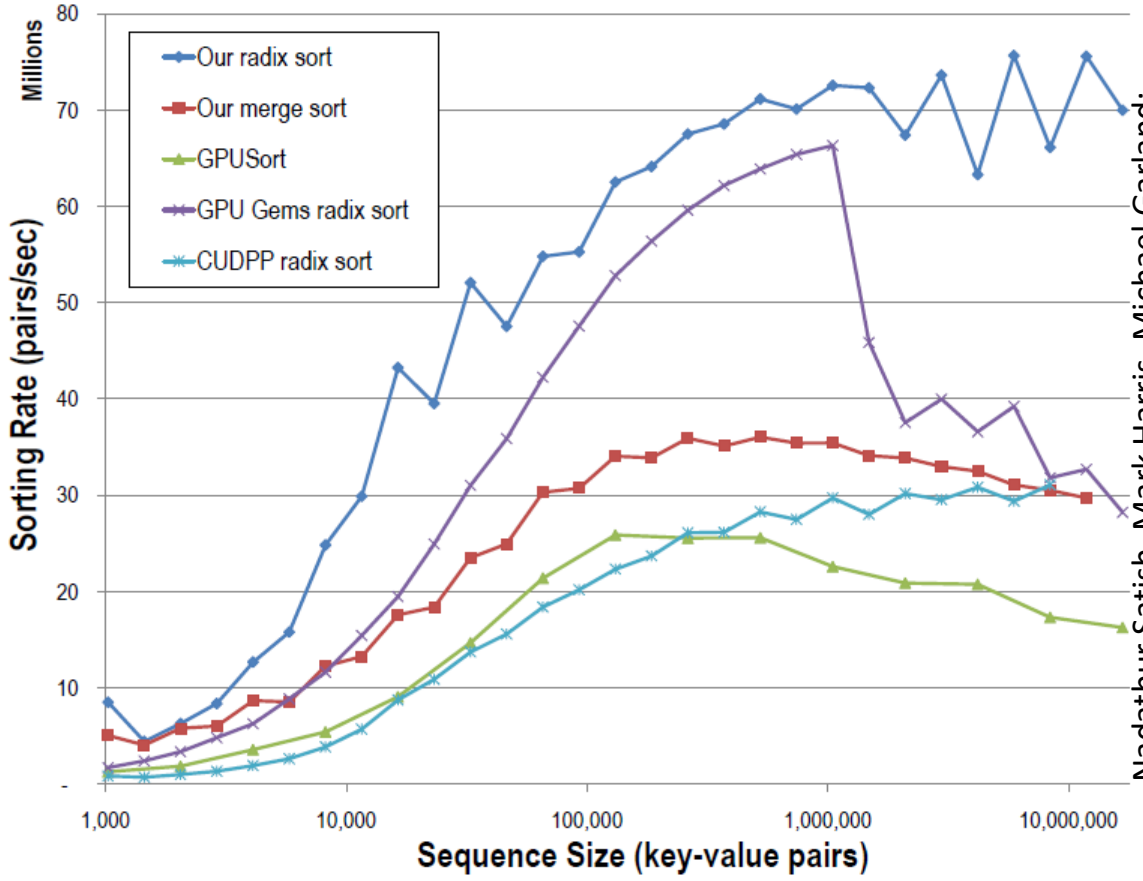
Every thread wants to spawn a different number of child threads.
->What's the parent for each child?

```
__global__ void thisstep(uint *numproc, ...){
  ...
  numproc[linid] = newprocessors();
}
__global__ void map(uint *numproc, uint *mapping){
  mapping[numproc[linid]] = 1;
}
__global__ void nextstep(uint *mapping, ...){
  uint myelement = mapping[linid];
  ...
}
thisstep<<<x,y>>>(numproc, ...);
uint sum = scanExlusive(numproc);
mapping = gpuMallocNSet<uint>(sum,0);
map<<<x,y>>>(numproc, mapping);
scanInclusive(mapping);
nextstep<<<sum/bs,bs>>>(mapping, ...);
```

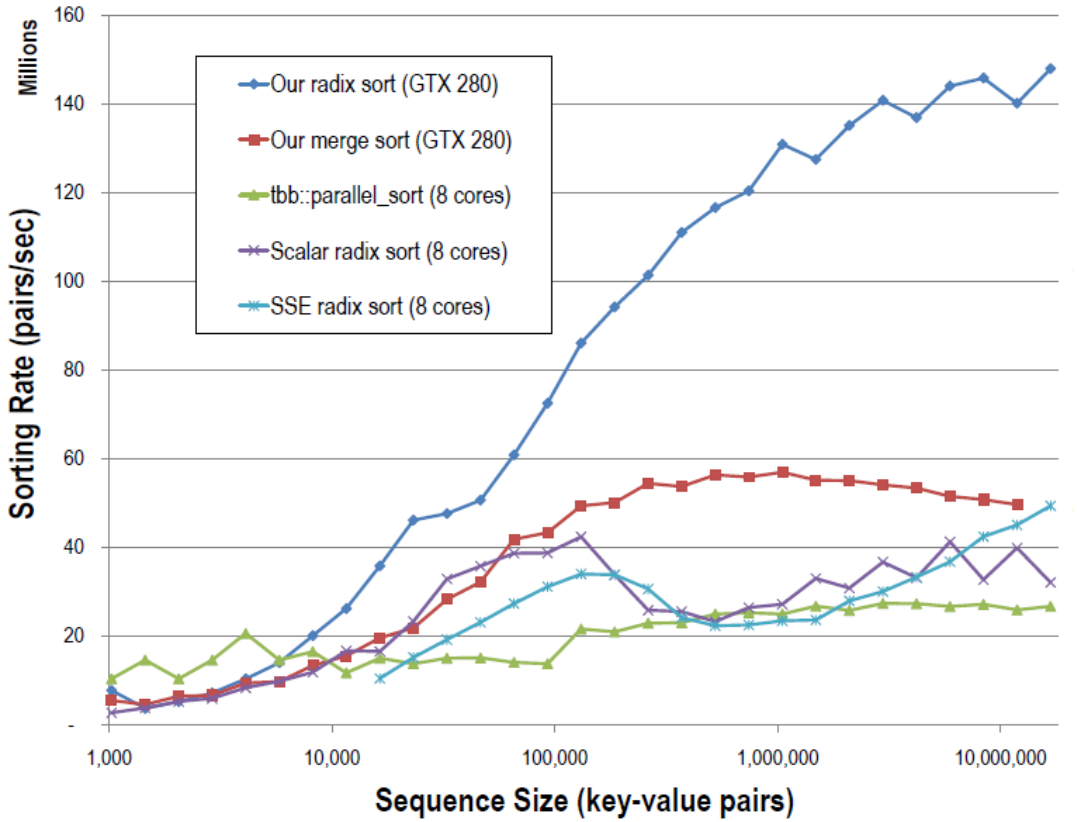
```
numproc:[2,1,3,1]
     7 = [0,2,3,6]
mapping:[0,0,0,0,0,0,0]
mapping:[1,0,1,1,0,0,1]
mapping:[1,1,2,3,3,3,4]
```

Sort

- Often required, often used
- Need more than 10k keys to be faster than CPU
- For a high number of keys, radix sort seems to perform best
- For a low number of keys (< 10k), bitonic merge sort might be a better choice
- For small amounts of data, insertion sort might even work well
- → experiment

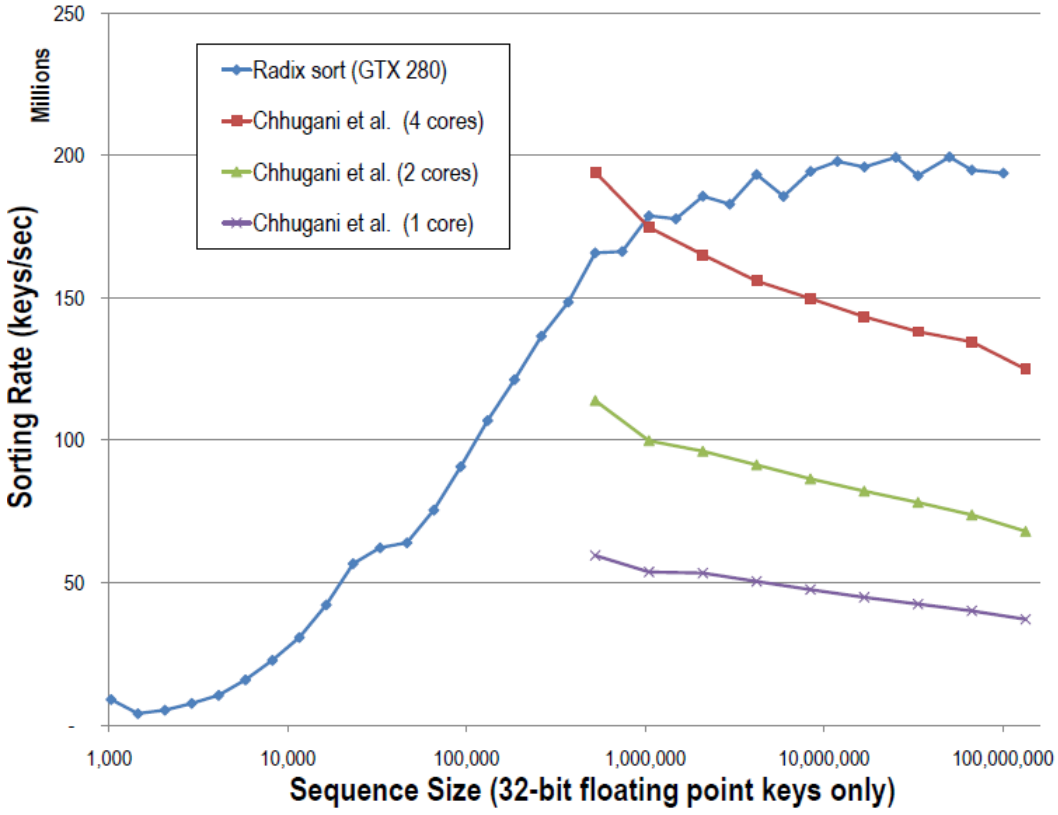


Nadathur Satish, Mark Harris, Michael Garland: Designing Efficient Sorting Algorithms for Manycore GPUs



(a) 8-core Clovertown (key-value pairs)

Nadathur Satish, Mark Harris, Michael Garland: Designing Efficient Sorting Algorithms for Manycore GPUs



(b) 4-core Yorkfield (float keys only)

Nadathur Satish, Mark Harris, Michael Garland: Designing Efficient Sorting Algorithms for Manycore GPUs

ATOMIC OPERATIONS

Mutual Exclusion

- CPU mutex strategies can hardly be used
- Locks not feasible with thousands of threads
- Avoid operations on same memory
- Use data convergence points as sparsely as possible

→ rely on atomic operations

(alternative: reduction)

Atomic Ops

- Atomic ops are supported for shared and global memory
- `atomicAdd`, `atomicSub`, `atomicExch`, `atomicMin`, `atomicMax`, `atomicInc`, `atomicDec`, `atomicCAS`, `atomicAnd`, `atomicOr`, `atomicXor`
- Some only available int/uint data types
- One of the most important tools for parallel programming

Atomic Ops cont'd

serialized when multiple threads access the same word in parallel

→ Performance loss (timings: GTX 980Ti)

global memory, 8 blocks		
words	atomic	non-atomic
1	9226.9	5784.4
2	6974.8	5973.7
4	3484.5	6264.9
8	3485.9	6521.3
16	2613.7	6414.6
32	2614.2	5116.4
64	2613.7	5413.6
128	1306.9	2839.0
256	1306.9	1533.4
512	655.4	1250.9
1024	246.0	505.1

Atomic Ops Example

```
__device__ uint front, back;
__device__ T ringbuffer[RingSize];

__device__ void push_back(T data)
{
  uint spot = atomicInc(&back, RingSize-1);
  ringbuffer[spot] = data;
}

__device__ T pop_front()
{
  uint spot = atomicInc(&front, RingSize-1);
  return ringbuffer[spot];
}
```

simplyfied ringbuffer

Mutex?

```
__device__ uint mutex = 0;

__device__ void acquire()
{
  while(atomicCAS(&mutex,0,1) == 1);
  /*mutex acquired*/
}

__device__ void release()
{
  atomicCAS(&mutex,1,0);
}
```

hardware scheduler tries to collect warp here!

BARRIERS AND VOTES

Synchronization

- Synchronize all threads in a block (device code)

```
void __syncthreads()
```

- Synchronize on kernel launch (host code)

```
void cudaDeviceSynchronize()
```

Synchronization with Voting (CC>=2.0)

- Get additional information when synchronizing

```
int __syncthreads_count(int predicate)
```

number of threads with predicate != 0

```
int __syncthreads_and(int predicate)
```

logical and on predicate of all threads

```
int __syncthreads_or(int predicate)
```

logical or on predicate of all threads

Atomic Ops and Barriers

```
__device__ uint front, back;
__device__ T ringbuffer[RingSize];

__device__ void push_back(T data)
{
  uint spot = atomicInc(&back, RingSize);
  ringbuffer[spot] = data;
}

__device__ T pop_front()
{
  uint spot = atomicInc(&front, RingSize);
  return ringbuffer[spot];
}
```

simplyfied ringbuffer

Barriers / Fences

- Make sure that data is available before continuing

```
__device__ uint front, back;
__device__ T ringbuffer[RingSize];
__device__ uint ringbuffer_flags[RingSize];

__device__ void push_back(T data)
{
  uint spot = atomicInc(&back, RingSize);
  ringbuffer[spot] = data;
  __threadfence();
  ringbuffer_flags[spot] = Ready;
}
```

make sure data is visible

flag as being ready

simplyfied ringbuffer 2

Warp Votes

- Fast votes within warps

```
int __all(int predicate)
```

non-zero if all threads' predicates are non-zero (CC>=1.2)

```
int __any(int predicate)
```

non-zero if any thread's predicate is non-zero (CC>=1.2)

```
int __ballot(int predicate)
```

n^{th} bit is set if n^{th} thread's predicate is non-zero (CC>=2.0)

Votes cont'd

```
do
{
  ...
  bool run = shouldRun();
} while(__any(run));
```

Example 1: state-based loop

```
__device__ void myfunction(...)
{
  uint bres = __ballot(1);
  uint activeThreads = __popc(bres);
  ...
}
```

Example 2: active threads

CPU-GPU INTERACTION

Memcopy (GTX 680)

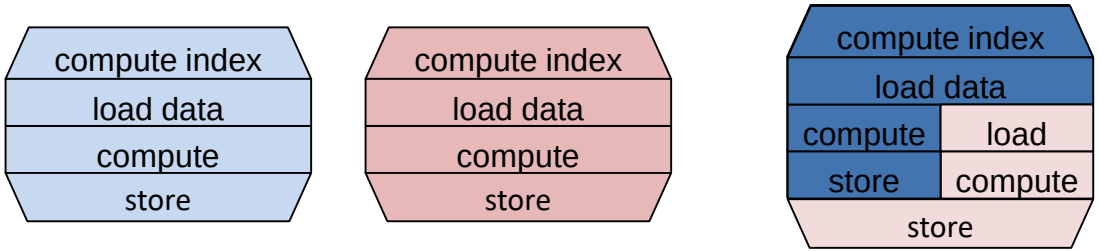
- page-able (standard):
  - host-to-device: 3.1GB/s
  - device-to-host: 3.2GB/s
- pinned:
  - host-to-device: 5.8GB/s
  - device-to-host: 6.1GB/s
  - asynchronous copy with CPU
  - asynchronous copy with GPU
  - use cudaHostAlloc /cudaHostFree
  - can not use too much pinned memory…
- One big memcopy is always faster than many small ones
- device-to-device copy: 150-170 GB/s

Zero-copy

- Pinned memory can be mapped into device memory space
- Data is transmitted on demand
- Can be faster if …
  - … data is read and written only once
  - … only a few data words are read
  - … there is a lot of arithmetic complexity hiding the memory transfer
  - … you are on a mobile device which shared the RAM
  - … data access is coalesced
- Could be needed if
  - … you want to communicate with currently running kernel

Kernel fusion

- Kernel launching and index calculations can create a serious overhead
- → Combine multiple kernels if possible
  - Identical grid and block layout
  - Successive steps on same data
  - Similar shared memory requirements
  - Similar register needs



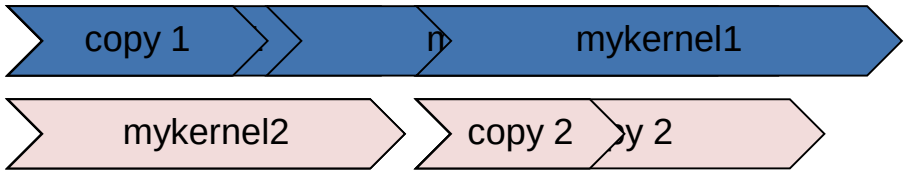
Asynchronous Execution

- Default Interface:
  - Kernel launches are asynchronous with CPU
  - Memcopy blocks calling CPU thread
  - CUDA calls are serialized by the driver
- Streams and asynchronous functions:
  - Memcopy asynchronous with CPU
  - Execute kernel and memcopy concurrently
  - Execute multiple kernels concurrently (CC 3.5 and up)
  - Pinned memory for memcpy
  - Different non-0 streams

Asynchronous Execution cont'd

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);


cudaMemcpyAsync(d_data1, h_data1, H2D, stream1);
mykernel1<<< 64,128,0,stream1>>>(arg2, arg3);
mykernel2<<<128,128,0,stream2>>>(arg0, d_loc2);
cudaMemcpyAsync(h_data2, d_loc2, D2H, stream2);
```



CONTROL FLOW

Stack

- Usually all function calls are inlined
- Fermi (cc >= 2.0) offers a stack
- Function pointers supported
- Recursion supported
- virtual functions supported
- Call costs 200-500 cycles
  → avoid if possible

```
cudaDeviceSetLimit(cudaLimitStackSize, <size>)
```

Loop unrolling

- Loop condition evaluation can form a serious overhead
- Make sure that loops can be unrolled if possible
- Test if loops are really unrolled:

```
#pragma unroll
for(uint i = 0; i < 10; ++i)
{
  ...
  //this would be automatically
  unrolled by the compiler anyway
}
```

Warp Shuffle CC >= 3.0

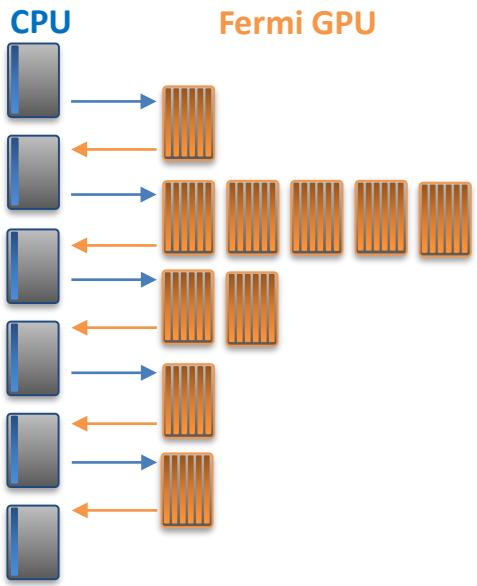
- Exchange a variable between threads within a warp without shared memory
- Can only read values from participating threads

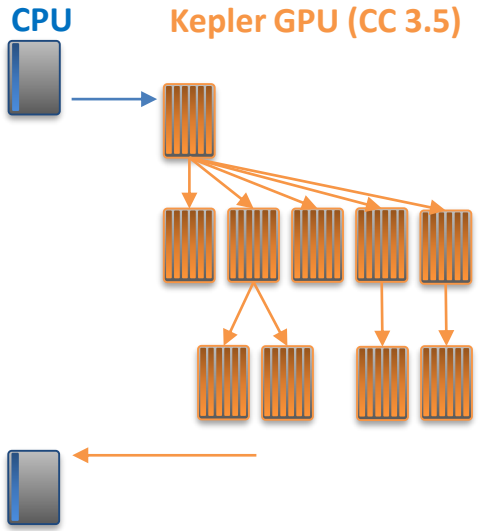
```
int __shfl(int var, int srcLane, int width = Warpsize)
```

- Special shuffle functions with deltas and XOR
- Enables faster reduction methods
- Reduces shared memory pressure

Dynamic Paralellism

GPU adapts to data, dynamically launches new threads



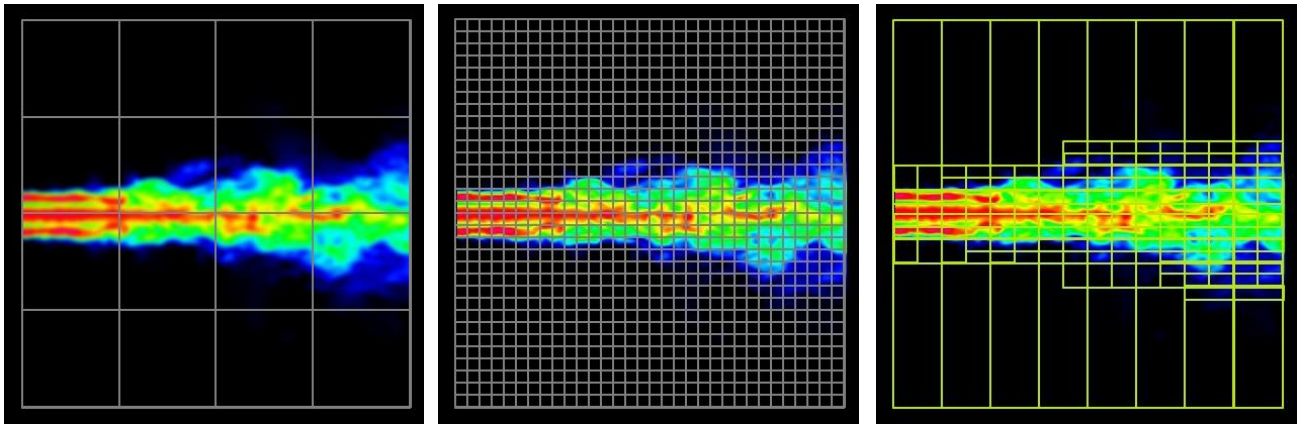


Dynamic Paralellism

too coarse

too fine

dynamic



Dynamic Paralellism

```
__global__ void kernel1(..);
__global__ void kernel2(..);
__global__ void main_kernel(..)
{
  int mydata = complexComputation(..);
  int state = 0;
  if(mydata == STATE_1)
  {
    kernel1<<<1,512>>>(globalPointer, ..);
    cudaDeviceSynchronize();
    if(globalPointer[i] == result1)
    {
      kernel2<<<4,32>>>(globalPointer, ..);
      cudaDeviceSynchronize();
      if(globalPointer[i] == result2)
        state = 1;
    }
  }
  __syncthreads_count(state);
}
```

each thread starts a kernel into the same stream

each thread synchronizes to previous kernel launches

data can only be exchanged via global memory

Due to the cudaDeviceSynchronize the entire block waits until all child kernels have been executed; if no cudaDeviceSynchronize was called at least on thread waits until all child kernels are executed

CUDA LIBRARIES

Nvidia Performance Primitives

- Nvidia Performance Primitives (NPP)
- Image and Video Processing
- Set of functions to do:
  - thresholding, morphological operations
  - convolution
  - min, max, mean, median, sum etc.
  - add, sub, mul, bit-operators etc.
  - graph cut
  - histogram
  - color space and data format conversions
  - discrete cosine transformation
  - geometric transformations

Parallel Primitives Library

- CUDA Data Parallel Primitives Library (CUDPP)
- NVIDIA Thrust
- Prefix sum, scan, sort

- CUDA Fast Fourier Transform (CUFFT)

- OpenCV GPU-extension
  - can do a bit more than NPP
    e.g. more filtering and segmentation algorithms, feature detection, camera calibration and stereo
  - uses OpenCV data types (in contrast to NPP)

Linear Algebra Libraries

- CUDA Basic Linear Algebra Subprograms (CUBLAS)
- CUSPARSE for sparse matrices
- Only very basic vector and matrix functions:
  - multiply vectors with matrices or scalars
  - transpose, convert from and to sparse
  - dot products, combined multiply-add etc.
- More complex algorithms (eigenvalues, matrix decompositions, equation solvers etc.) not included!

CUDA Libraries

- All of these libraries are C++ libraries!
  GPU code linking is quite new and is inefficient (function call).
  Libraries either take host data or pointer to device data, will do their own kernel launches.
- Usually no parallelization across multiple GPUs.
  You need to add device-switching etc. on your own.

CUDA AND OPENGL

Graphics

- CUDA offers great flexibility
- No rigid pipeline, code as you wish

- However:
  - not easy to beat the hardware graphics pipeline (pipeline is highly optimized)
  - hardware rasterizer & ROPs

Rendering paradigms

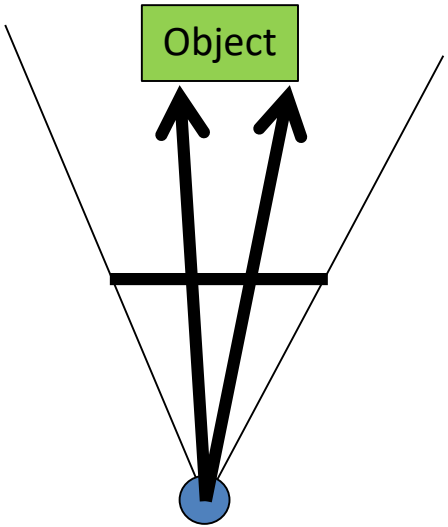
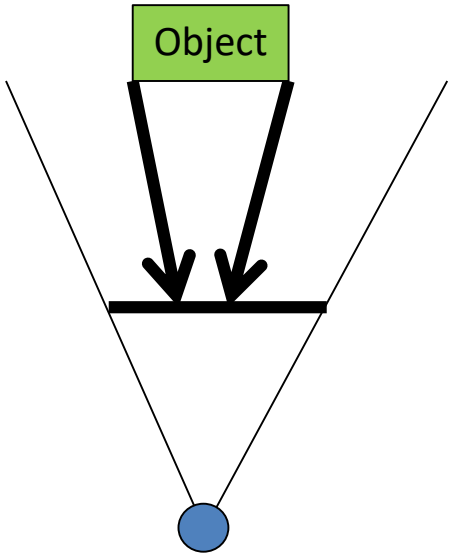


Image Ordered
fits CUDA well



Object Ordered
most often easier with OpenGL

OpenGL Interoperability

- Reading and writing OpenGL resources (vertex and pixel buffers, textures) has at least a small overhead
- If you have an OpenGL-based pipeline and only want to replace some minor stages with CUDA, consider using shaders instead

Post-processing a Rendered Image

1. Render scene in OpenGL to textures
2. Map textures to CUDA: cudaGraphicsGLRegisterImage()
3. Process the buffer
4. Unmap the buffer
5. Render a screen sized quad/triangle (cannot directy map framebuffer)

Processing Vertices

1. Map VBO: cudaGraphicsGLRegisterBuffer()
2. Process the vertex buffer
3. Unmap the resource
4. Render VBO in OpenGL

MULTI GPU

Multi-GPU

- No automatic multi-GPU, you have to do it manually
- cudaSetDevice() selects the GPU that receives subsequent commands
- Earlier CUDA-versions: you needed one CPU-thread for each GPU

Multi-GPU Execution

- Today this is ok (single-threaded):

```
for(int i = 0; i < numGPUs; i++)
{
  cudaSetDevice(i);
  launch_kernel<<<....>>>(..);
}
```

- Don‘t forget to use async memcopy to avoid serializing GPUs via the host
- Of course, use cudaDeviceSynchronize() only before and after the parallel execution

Multi-GPU Memory Transfers

- If you want to exchange data between GPUs: copy through host memory explicitly:
  1) copy from device 1 to CPU
  2) copy from CPU to device 2

- Slow because data goes up and down the bus
- Slow because CPU is synced to GPUs
- Annoying to code

Multi-GPU Memory Transfers

- Unified address space for all GPUs and CPU
- You just copy between pointers, the device figures out where the data is located
- If it‘s GPU-to-GPU it can copies data directly over PCIe
- Requirements:
  - Linux or Windows TCC (Tesla Compute Cluster) drivers
  - 64 bit device code

OPENCL VS CUDA

OpenCL vs CUDA

OpenCL

- support for AMD and NVIDIA graphics cards and CPUs
- JIT compilation
- GPU kernel debugging only for AMD cards (gDEBugger)
- emulation debugging
- complex host interface
- differences between vendors
- built on top of C99
- code restrictions (e.g. pointer to pointer)

CUDA

- NVIDIA cards only
- kernel debugging using visual studio / gdb / nsight
- JIT compilation based on ptx possible
- simple host interface
- latest GPU features
- C++ features like classes and templates
- Standard libraries (CUDPP, cuFFT, cuBLAS, cuRAND…)
- templates!
- templates!

AMD vs NVIDIA
Current Generation

AMD R9 290 „Volcanic Islands“

- 44 compute units (CU)
  x (4 x 16 SIMD)
  2816 cores
- 947 MHz core clock
- 1250 MHz memory clock
- 4 memory controllers
- 320 GB/s memory bandwidth
- 16 KB L1 cache per CU
- 1MB L2 cache
- 64 KB shared memory
- 28 nm process
- 290W max power

NVIDIA GTX Titan Black „Kepler“

- 15 streaming multiprocessors (SMX)
  x 192 CUDA cores
  2880 cores
- 980 MHz core clock
- 1750 MHz memory clock
- 4 memory controllers
- 336 GB/s memory bandwidth
- 16/32/48 KB L1 cache per SMX
- 1536 KB L2 cache
- 48/32/16 KB shared memory
- 28 nm process
- 250W max power

AMD vs NVIDIA
Compute Architecture

AMD Accelerated Parallel Processing

- wavefronts: size 64 (SIMD 16)
- VLIW → SIMD
  Graphics Core Next (GCN)
- DMA, zero-copy

NVIDIA CUDA

- warps: size 32
- SIMT / SIMD
- DMA, zero-copy
- stack
- function pointer
- recursion
- unified address model
- direct access to other GPU‘s memory
- large set of special instructions

What to use?

OpenCL

- wide releases
- for simpler programs
- use AMD cards and tools
- more development time

CUDA

- prototyping
- known target architecture
- complex programs
- math libraries / primitives
- special instructions
- LLVM compiler → open

ALGORITHMS

Algorithm

- High degree of parallelism
  - Thousands of threads
  - In multiple stages
  - Data parallelism
  - Fixed parallelism in each stage
- High arithmetic load
- Groups of threads work coherently
  - little branching
- Little need for reductions
- Similar execution time for all threads
- Few dependencies between successive kernels

Well fitting algorithms 1

- Complex image filtering: 1 000 000 threads
  - per thread: a few inputs, one output
  - well aligned memory access
  - no dependencies
- Computational Fluid Dynamics: 10 000 threads
  - many iterations of complex math
  - well aligned memory access
- Protein sequencing: 8 000 threads
  - compute intensive Viterbi algorithm
  - well aligned memory access
  - threads finish at the same time

Well fitting algorithms 2

- Finance risk management: 100 000 threads
  - parallel Monte-Carlo simulations
  - independent execution paths
  - parallel reduction
- Weather Prediction: 1 000 000 threads
  - many iterations of complex math (2000+ ops)
  - well aligned memory access
- MRI reconstruction: 1 000 000 threads
  - use of FFT
  - many operations
  - well aligned memory access

Not so well fitting algorithms

- Volume rendering of unstructured grids: 1 000 000 threads
  - divergent paths based on hit cell
  - unstructured memory access patterns
  - varying execution durations

- Octree-based mesh simplification: 1 – 10 000 000 threads
  - varying parallelism throughout the algorithm
  - unordered writes, unknown dependencies between threads
  - low parallelism for low depths
  - execution configuration of next level depends on results from last
  - mapping threads to nodes complex

Questions