GPU programming:
Unified memory models and more

Sylvain Collange
Inria Rennes – Bretagne Atlantique
sylvain.collange@inria.fr
OK, I lied

- You were told
  - CPU and GPU have distinct memory spaces
  - Blocks cannot communicate
  - You need to synchronize threads inside a block
- This is the least-common denominator across all CUDA GPUs
- This is not true (any more). We now have:
  - Global and shared memory atomics
  - Warp-synchronous programming, warp vote, shuffle
  - Device-mapped, Unified virtual address space, Unified memory
  - Dynamic parallelism
Outline

- Device-mapped memory, Unified virtual address space, Unified memory
- Dynamic parallelism
- Other memory spaces: texture, constant
- Pitfall and fallacies
NVIDIA Compute capabilities

- Newer GPUs introduce additional features

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Tesla</th>
<th>Fermi</th>
<th>Kepler</th>
<th>Maxwell</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>G80</td>
<td>G92</td>
<td>GT200</td>
<td>GT21x</td>
</tr>
<tr>
<td></td>
<td>GF100</td>
<td>GF104</td>
<td>GK104</td>
<td>GK110</td>
</tr>
<tr>
<td></td>
<td>GM107</td>
<td>GM104</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
<td>1.1</td>
<td>1.3</td>
<td>1.2</td>
<td>2.0</td>
<td>2.1</td>
<td>3.0</td>
<td>3.5</td>
<td>5.0</td>
<td>5.2</td>
</tr>
</tbody>
</table>

- Compute capability means both
  - Set of features supported
    - Who can do more can do less: $x > y \rightarrow \text{CC } x \text{ includes CC } y$
  - Native instruction set
    - Not always backward-compatible
    - e.g. GPU of CC 2.0 cannot run binary for CC 1.3
Device-mapped memory

- The GPU can access host memory directly, if explicitly mapped in the device address space
- **Lower bandwidth** but **lower latency** than DMA transfers
  - Useful for light or sparse memory accesses
- Advantageous for integrated GPUs
- Still different **address spaces**
  Device address is different than host address

![Diagram of device-mapped memory](image-url)
Example

// Allocate device-mappable memory
float* a;
cudaHostAlloc((void**)&a, bytes, cudaHostAllocMapped);

// Get device-side pointer
float* da;
cudaHostGetDevicePointer((void**)&d_a, a, 0);

// Write using host pointer on CPU
a[0] = 42;

// Read/write using device pointer on GPU
mykernel<<<grid, block>>>(d_a);

// Wait for kernel completion
cudaDeviceSynchronize();

// Read using host pointer
printf("a=%f\n", a[0]);

// Free memory
cudaFreeHost(a)
Unified virtual address space (UVA)

- Coordinate allocation in CPU and GPU address space
  - Can tell if an address is on the device, host or both
- cudaHostAlloc returns an address valid in both host and device space
- No need to specify direction in cudaMemcpy
- Caveat: not a true unified address space
  - Having an address does not mean you can access the data it points to
  - Memory allocated with malloc still cannot be accessed by GPU
- Requirements: CUDA $\geq$ 4.0, 64-bit OS, CC $\geq$ 2.0
  - Not available on Windows Vista/7 under WDDM
Unified memory: the real thing

- Allocate memory using `cudaMallocManaged`
  - The CUDA runtime will take care of the transfers
  - No need to call `cudaMemcpy` any more
  - Behaves as if you had a single memory space

- Suboptimal performance: software-managed coherency
  - Still call `cudaMemcpy` when you can

- Requires CUDA ≥ 6.0, CC ≥ 3.0 (preferably 5.x), 64-bit Linux or Windows
VectorAdd using unified memory

```c
int main()
{
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    float *A, *B, *C;
    cudaMemcpyManaged((void **)&A, size);
    cudaMemcpyManaged((void **)&B, size);
    cudaMemcpyManaged((void **)&C, size);
    Initialize(A, B);

    int blocks = numElements;
    vectorAdd2<<<blocks, 1>>>(A, B, C);
    cudaDeviceSynchronize();
    Display(C);

    cudaFree(A);
    cudaFree(B);
    cudaFree(C);
}
```

- Can I replace cudaMemcpyManaged by cudaMemcpyHostAlloc (with UVA?)
  - What is the difference?
Multi-GPU programming

- What if I want to use multiple GPUs on the same machine?
- `cudaGetDeviceCount`, `cudaGetDeviceProperties` enumerate devices
- `cudaSetDevice(i)` selects the current device
  - All following CUDA calls in the thread will concern this device
- Streams are associated with devices
  - But `cudaStreamWaitEvent` can synchronize with events on other GPU: allow inter-GPU synchronization
- Host memory accessible from multiple devices: `cudaHostAlloc(..., cudaHostAllocPortable)`
  - Then call `cudaHostGetDevicePointer` for each GPU
Peer-to-peer memory copy/access

- Transfer memory between GPUs
  - cudaMemcpyDeviceToDevice will not work. Why?
  - Without UVA
    cudaMemcpyPeer()
    cudaMemcpyPeerAsync()
  - With UVA:
    just plain cudaMemcpy
- Direct access to other GPU's memory
  - Check result of cudaDeviceCanAccessPeer
  - Call cudaDeviceEnablePeerAccess from accessing GPU
Recap

- We have too many mallocs...
  - cudaMalloc
  - cudaMallocHost
  - cudaHostAlloc(..., cudaHostAlloc{Portable|Mapped})
  - cudaMallocManaged

- And too many memcpys
  - cudaMemcpy(dest, src, cudaMemcpy{HostToDevice, DeviceToHost, DeviceToDevice, Default})
  - cudaMemcpyAsync
  - cudaMemcpyPeer[Async]

- Quizz: do you remember what they do and when we should use them?
Recap: evolution of the memory model

New features: going away from the split memory model

- Device-mapped: GPU can map and access CPU memory
  - Lower bandwidth than DMA transfers
  - Higher latency than GPU memory access

- Uniform virtual addressing (CC 2.0):
  synchronize memory space between CPU and GPU
  - Addresses are unique across the system
  - No need to specify direction in cudaMemcpy

- Unified memory (CC 3.x):
  both CPU and GPU share a managed memory space
  - Driver manages transfers automatically
  - Only for memory allocated as managed
  - Unmanaged + cudaMemcpy still useful for optimized transfers
Outline

- Device-mapped memory, Unified virtual address space, Unified memory
- Dynamic parallelism
- Other memory spaces: texture, constant
- Pitfall and fallacies
Dynamic parallelism

- Kernels can launch kernels

```
kernel<<<blocks, threads, smem, stream>>>(arguments);
```

- Same syntax as host code
  - Use streams for parallel kernel calls from the same block

- Applications
  - Traversal of graphs and irregular structures
  - Multi-resolution finite element analysis
Texture memory

- Primarily made for graphics
- Optimized for 2D accesses
  - Takes advantage of locality in space
- Not coherent caches: usually read-only
- Initialization through specific host functions
  - cudaCreateTextureObject(), cudaBindTexture2D()...
- Access through specific device functions
  - tex1D(), tex2D(), tex3D()...
Bilinear texture filtering

- Fixed-function unit in the texture read path
- Interpolate color of **Pixels** from **Texels**

\[ T[i,j] \rightarrow T[i,j+1] \rightarrow \alpha \rightarrow \beta \rightarrow y \]

- Alignment (FP->FX)
- Bilinear interpolation FX
- MAC FP*FX+FP->FP

**Applications:** graphics, image processing
- Can implement 2D lookup tables with interpolation
Constant memory

- Array or variable declared as __const__
- Initialized in place or from host code, then read-only
- Used automatically by the compiler for compile-time constants in code
- Single-ported memory: good performance when all threads in a warp access the same word (broadcast)
  - Similar to shared memory with 1 bank
  - Generally not worth using with divergent accesses
- Being phased out in CUDA, but may appear in existing code
Cached constants in global memory

- Non-coherent cache for constants on newer NVIDIA GPU architectures
- Complements or replace the constant memory space
- Special load instruction in PTX or assembly
- Two ways to use it at the C for CUDA level:
  - Automatic: be sure to declare all pointers to constant data as const restrict. The compiler may be able to prove accesses are read-only
  - Manual: force use of the cache using the __ldg() intrinsic
Conclusion: trends

- GPU: rapidly evolving hardware and software
- Going towards CPU-GPU tighter coupling
  - On-chip integration
  - Shared physical memory
  - Shared virtual memory space
- Most development is going into mobile
  - Nvidia: Kepler GPUs in Tegra 4 support CUDA
  - Many GPUs supporting OpenCL: ARM Mali, Qualcomm Adreno, Imagination Technologies PowerVR Rogue...
- Still much work to do at the operating system level
  - GPU currently a second-class citizen
  - Need to move from CPU+devices model to heterogeneous compute model
References and further reading/watching

- CUDA C Programming Guide
Pitfalls and fallacies
GPUs are 100x faster than CPUs
GPUs are 100x faster than CPUs

- Wrong
  - The gap in peak performance (compute and memory) is 10x
  - In practice, 5x to 8x against optimized CPU code

- Right: you can get a 100x speedup when porting an application to GPU
  - You get 10x because of the higher GPU throughput
  - ... and 10x more because you parallelized and optimized the application
    But spending the same effort on optimizing for CPU would also yield 10x improvement
      - Modern CPUs are highly parallel too

- Right: if you use GPU hardware features not available on CPUs
  - Texture filtering, elementary functions (exp, log, sin, cos)...

GPUs have thousands of cores
GPUs have thousands of cores

- Official definition of CUDA Core: one execution unit
  - Right: GPUs have thousands of cores
  - Using this metric, CPUs have hundreds of cores!

- The closest to a CPU core is a GPU SM
  - Wrong: 15 to 20 SMs “only” on a GPU
  - Wide SIMD units inside
    But just 2x to 4x wider than on CPU

- Often-overlooked aspect:
  GPUs have 10s of thousands of threads
  - But: CPU thread ~ GPU warp
  - Still thousands of warps
I can avoid divergence with predication
I can avoid divergence with predication

Do not confuse predication with prediction!

- In a superscalar CPU, branches are predicted
  - Cost of misprediction ~ pipeline depth independent from length of mispredicted path
  - Modern branch predictors are really good at finding correlations
  - Inefficient on short ifs governed by random conditions
    Predication might help in this case

- In a GPU, instructions are predicated
  - Cost of divergence ~ divergent section size
  - GPU divergence management has very low overhead
  - Inefficient on long ifs/loops governed by divergent conditions
    Predication does not help in this case