Programmierung paralleler und verteilter Systeme

GPU Computing with OpenCL

Frank Feinbube

Operating Systems and Middleware
Prof. Dr. Andreas Polze
Agenda / Quicklinks

- Motivation
- History of GPU Computing
- Programming Model
- Examples
- Development Support
- Hardware Characteristics
- Performance Tuning
- Further Readings
The Power of GPU Compute Devices

Fluids

NBody

Using CUDA device [01]: GeForce GTX 275
Sorting 1048576 32-bit unsigned int keys and values
radixSort, Throughput = 74.6231 MElements/s, Time = 0.01405 s, Size = 1048576 elements, NumDevsUsed = 1, Workgroup = 256
PASSED
Wide Variety of Application Domains

Research
Medical
Video and Photo
Energy
Finance
Military

Statistical constraints on binary black hole inspirations
BMC Bioinformatics
Accelerating the Smith-Waterman Algorithm using graphics processing units
Graphic processors to speed-up simulations for the...

Cmatch: Fast Exact String Matching on the GPU
Quantitative Risk Analysis and Algorithmic Trading
Harvesting graphics power for Astrophysics (GraCCA)

Quantum Chemistry Two-Electron Integral Evolution
Accelerating Statistical Static Timing Analysis
Distributed Password Recovery
Folding@home

http://www.nvidia.com/object/tesla_testimonials.html
Why GPU Compute Devices?
Short Term View: Cheap Performance

Performance

![Graph showing execution time in milliseconds vs. problem size (number of Sudoku places) for Intel E8500 CPU, AMD R800 GPU, and NVIDIA GT200 GPU. Lower means faster.]

Energy / Price

- Cheap to buy and to maintain
- GFLOPS per watt: Fermi 1.5 / Keppler 5 / Maxwell 15
Why GPU Compute Devices?
Middle Term View: More Performance
Why GPU Compute Devices? 
Middle Term View: More Performance
Why GPU Compute Devices?
Long Term View: Hybrid Computing

Dealing with massively multi-core:
- New architectures are evaluated (Intel, IBM, HP,...)
- Accelerators that accompany common general purpose CPUs (Hybrid Systems)

Hybrid Systems
- **GPU Compute Devices:**
  High Performance Computing (3 of top 5 supercomputers are GPU-based!), Business Servers, Home/Desktop Computers, Mobile and Embedded Systems
- **Special-Purpose Accelerators:**
  (de)compression, XML parsing, (en|de)cryption, regular expression matching
History of GPU Computing

- **CPU**
  - Evolving toward throughput computing
  - Motivated by energy-efficient performance

- **GPU**
  - Evolving toward general-purpose computing
  - Motivated by higher quality graphics and data-parallel programming

**Throughput Performance**

- **Multi-threading**
- **Multi-core**
- **Many Core**

**Programmability**

- **Larrabee**
  - Fully Programmable
  - Partially Programmable
  - Fixed Function
# History of GPU Computing

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Function Graphic Pipelines</td>
<td>• 1980s-1990s; configurable, not programmable; first APIs (DirectX, OpenGL); Vertex Processing</td>
</tr>
<tr>
<td>Programmable Real-Time Graphics</td>
<td>• Since 2001: APIs for Vertex Shading, Pixel Shading and access to texture; DirectX9</td>
</tr>
<tr>
<td>Unified Graphics and Computing Processors</td>
<td>• 2006: NVIDIA's G80; unified processors arrays; three programmable shading stages; DirectX10</td>
</tr>
<tr>
<td>General Purpose GPU (GPGPU)</td>
<td>• compute problem as native graphic operations; algorithms as shaders; data in textures</td>
</tr>
<tr>
<td>GPU Computing</td>
<td>• Programming CUDA; shaders programmable; load and store instructions; barriers; atomics</td>
</tr>
</tbody>
</table>
Open Compute Language (OpenCL)

- **AMD**
  - Merged, needed commonality across products

- **ATI**
  - GPU vendor – wants to steal market share from CPU

- **NVIDIA**
  - CPU vendor – wants to steal market share from GPU

- **Intel**
  - Was tired of recoding for many core, GPUs. Pushed vendors to standardize.

- **Apple**

Khronos Compute Group formed

- Wrote a draft straw man API

- OpenCL

- **Ericsson**

- **Sony**

- **Texas Instruments**

- **Blizzard**

- **IBM**

- **Nokia**
Open Compute Language (OpenCL)

- Hardware vendors, system OEMs, middleware vendors, application developers
- OpenCL became an important standard “on release” by virtue of the market coverage of the companies behind it.
- OpenCL implementations already exist for AMD, NVIDIA, Intel, IBM, ARM, ...

- Use all computational resources in system
  - Program GPUs, CPUs, and other processors as peers
  - Efficient C-based parallel programming model
  - Abstract the specifics of underlying hardware

- Abstraction is **low-level, high-performance but device-portable**
  - Approachable – but primarily targeted at expert developers
  - Ecosystem foundation – no middleware or “convenience” functions

- Implementable on a range of embedded, desktop, and server systems
  - HPC, desktop, and handheld profiles in one specification
Programming Models

AMD: ATI Stream SDK
- Today using OpenCL

NVIDIA: Common Unified Device Architecture
- CUDA C/C++ compiler, libraries, runtime
- Mature: literature, examples, tool, development support

Khronos Group: OpenCL
Open standard for portable, parallel programming of heterogeneous parallel computing CPUs, GPUs, and other processors
OpenCL exposes CPUs, GPUs, and other Accelerators as “devices”

- Each “device” contains one or more “compute units”, i.e. cores, SMs, ...
- Each “compute unit” contains one or more SIMD “processing elements”
OpenCL execution model ... execute a kernel at each point in a problem domain.

**Traditional loops**

```c
void trad_mul(int n,
              const float *a,
              const float *b,
              float *c)
{
    int i;
    for (i=0; i<n; i++)
        c[i] = a[i] * b[i];
}
```

**Data Parallel OpenCL**

```c
kernel void
dp_mul(global const float *a,
        global const float *b,
        global float *c)
{
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
} // execute over "n" work-items
```

E.g., process a 1024 x 1024 image with one kernel invocation per pixel or 1024 x 1024 = 1,048,576 kernel executions
Data Parallelism versus Task Parallelism

Data Parallelism

Task 1

Task 1

Task 1

Task 1

Task 1

Task 1

Input Data

Parallel Processing

Result Data

Aggregation Task

Task Parallelism

Task 1

Task 2

Task 3

Aggregation Task

OpenCL is designed for SIMD / SPMD approaches

Programmierung paralleler und verteilter Systeme | GPU Computing | FF2015
OpenCL Execution Model

- Parallel work is submitted to devices by launching kernels.
- Kernels run over global dimension index ranges (NDRange), broken up into "work groups", and "work items".
- Work items executing within the same work group can synchronize with each other with barriers or memory fences.
- Work items in different work groups can’t sync with each other, except by launching a new kernel.
An example of an NDRRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs.
An OpenCL kernel is executed by an array of work items.

- All work items run the same code (SPMD)
- Each work item has an index that it uses to compute memory addresses and make control decisions
Work Groups: Scalable Cooperation

Divide monolithic work item array into work groups

- Work items within a work group cooperate via **shared memory**, **atomic operations** and **barrier synchronization**
- Work items in different work groups cannot cooperate
OpenCL Memory Architecture

**Private**
- Per work-item

**Local**
- Shared within a workgroup

**Global/Constant**
- Visible to all workgroups

**Host Memory**
- On the CPU
Memory management is explicit: you must move data from host → global → local... and back

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Keyword</th>
<th>Description/Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Memory</td>
<td>__global</td>
<td>Shared by all work items; read/write; may be cached (modern GPU), else slow; huge</td>
</tr>
<tr>
<td>Private Memory</td>
<td>__private</td>
<td>For local variables; per work item; may be mapped onto global memory (Arrays on GPU)</td>
</tr>
<tr>
<td>Local Memory</td>
<td>__local</td>
<td>Shared between workitems of a work group; may be mapped onto global memory (not GPU), else fast; small</td>
</tr>
<tr>
<td>Constant Memory</td>
<td>__constant</td>
<td>Read-only, cached; add. special kind for GPUs: texture memory</td>
</tr>
</tbody>
</table>
OpenCL Work Item Code

A subset of ISO C99 - without some C99 features
- headers, function pointers, recursion, variable length arrays, and bit fields

A superset of ISO C99 with additions for
- Work-items and workgroups
- Vector types (2,4,8,16): endian safe, aligned at vector length
- Image types mapped to texture memory
- Synchronization
- Address space qualifiers

Also includes a large set of built-in functions for image manipulation, work-item manipulation, specialized math routines, vectors, etc.
OpenCL codes must be prepared to deal with much greater hardware diversity (features are optional and may not be supported on all devices) → compile code that is tailored according to the device configuration
OpenCL Execution Model

An OpenCL application runs on a host which submits work to the compute devices. Kernels are executed in contexts defined and manipulated by the host.

- **Work item**: the basic unit of work on an OpenCL device.
- **Kernel**: the code for a work item. Basically a C function
- **Program**: Collection of kernels and other functions (Analogous to a dynamic library)
- **Context**: The environment within which work-items executes ... includes devices and their memories and command queues.
- **Queue**: used to manage a device. (copy memory, start work item, ...) In-order vs. out-of-order execution
OpenCL Context

- Contains one or more devices
- OpenCL memory objects are associated with a context, not a specific device
- `clCreateBuffer()` is the main data object allocation function
  - error if an allocation is too large for any device in the context
- Each device needs its own work queue(s)
- Memory transfers are associated with a command queue (thus a specific device)
OpenCL Device Command Execution

- Command-queue - coordinates execution of kernels
  - Kernel execution commands
  - Memory commands: transfer or mapping of memory object data
  - Synchronization commands: constrains the order of commands
Kernel body is instantiated once for each work item; each getting an unique index

```c
__kernel void vec_add (...)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

» Code that actually executes on target devices
Vector Addition: Host Program

```c
#include <stdio.h>

// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
    CL_DEVICE_TYPE_GPU, NULL, NULL, NULL, NULL);

// get the list of GPU devices associated with context
clGetContextInfo(context, CL_CONTEXT_DEVICES, 0, NULL, &cb);
devices = malloc(cb);
clGetContextInfo(context, CL_CONTEXT_DEVICES, cb, devices, NULL);

// create a command-queue
cmd_queue = clCreateCommandQueue(context, devices[0], 0, NULL);

// allocate the buffer memory objects
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_floa) * n, &cb, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_floa) * n, &cb, NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(cl_floa) * n, NULL, NULL);

// build the program
err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

// create the kernel
kernel = clCreateKernel(program, "vec_add", NULL);

// set the args values
err = clSetKernelArg(kernel, 0, (void *) &memobjs[0], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 1, (void *) &memobjs[1], sizeof(cl_mem));
err |= clSetKernelArg(kernel, 2, (void *) &memobjs[2], sizeof(cl_mem));

// set work-item dimensions
global_work_size[0] = n;

// execute kernel
err = clEnqueueNDRangeKernel(cmd_queue, kernel, 1, NULL, global_work_size, NULL, 0, NULL, NULL);

// read output array
err = clEnqueueReadBuffer(cmd_queue, memobjs[2], CL_TRUE, 0, n * sizeof(cl_floa), dst, 0, NULL, NULL);
```

Define platform and queues

Define Memory objects

Create the program

Build the program

Create and setup kernel

Execute the kernel

Read results on the host

„standard“ overhead for an OpenCL program
OpenCL “Hello Device”

Java+OpenCL “Fractal Explorer”
Dynamic Parallelism: The Idea

Fixed Grid

Dynamic Grid
Dynamic Parallelism in Action: SpaceX

Dynamic Parallelism in Action: SpaceX

Dynamic Parallelism: Task Distribution

**GPU as Co-Processor**

**Autonomous, Dynamic Parallelism**
Dynamic Parallelism: Familiar Syntax

```c
void main() {
    float *data;
    do_stuff(data);

    A <<< ... >>> (data);
    B <<< ... >>> (data);
    C <<< ... >>> (data);
    cudaDeviceSynchronize();
    do_more_stuff(data);
}

__global__ void B(float *data)
{
    do_stuff(data);

    X <<< ... >>> (data);
    Y <<< ... >>> (data);
    Z <<< ... >>> (data);
    cudaDeviceSynchronize();
    do_more_stuff(data);
}
```
Dynamic Parallelism: Code Example

- **CUDA Runtime syntax & semantics**
- **Launch is per-thread**
- **Sync includes all launches by any thread in the block**
- **`cudaDeviceSynchronize()` does not imply `syncthreads()`**
- **Asynchronous launches only (note bug in program, here!)**

```c
__device__ float buf[1024];
__global__ void dynamic(float *data)
{
    int tid = threadIdx.x;
    if(tid % 2)
        buf[tid/2] = data[tid]+data[tid+1];
    __syncthreads();
    if(tid == 0) {
        launch<<<128, 256>>>(buf);
        cudaDeviceSynchronize();
    }
    __syncthreads();
    cudaMemcpyAsync(data, buf, 1024);
    cudaDeviceSynchronize();
}
```
Development Support

**Software development kits:** NVIDIA and AMD; Windows and Linux

**Special libraries:** AMD Core Math Library, BLAS and FFT libraries by NVIDIA, OpenNL for numerics and CULA for linear algebra; NVIDIA Performance Primitives library: a sink for common GPU accelerated algorithms

**Profiling and debugging tools:**

- NVIDIAAs Parallel Nsight for Microsoft Visual Studio
- AMDs ATI Stream Profiler
- AMDs Stream KernelAnalyzer:
  - displays GPU assembler code, detects execution bottlenecks
- gDEBugger (platform-independent)

Big knowledge bases with tutorials, examples, articles, show cases, and developer forums
Nsight

Programmierung paralleler und verteilter Systeme | GPU Computing | FF2015
Towards new Platforms

- JavaScript binding to OpenCL
- Heterogeneous Parallel Computing (CPUs + GPU) within Web Browsers
- Enables compute intense programs like physics engines, video editing...
- Currently only available with add-ons (Node.js, Firefox, WebKit)

Android installable client driver extension (ICD)
- Enables OpenCL implementations to be discovered and loaded as a shared object on Android systems.

PGI for multicore ARM processors
GPU Hardware in Detail

NVIDIA GF100 GPU
GT200 – early architecture

Simpler architecture, but same principles

Several Work Groups reside on one SM
- Amount depends on available resources (Shared Memory (=Local Memory in OpenCL), Registers)
- More Work Groups → better latency hiding
  - Latencies occur for memory accesses, pipelined floating-point arithmetic and branch instructions

Thread execution in “Warps” (called “wavefronts” on AMD)
- Native execution size (32 Threads for NVIDIA)
- Zero-Overhead Thread Scheduling: If one warp stalls (accesses memory) next warp is selected for execution
Warp Execution Example

Application creates 200,000 „Tasks“
→ Global Work Group Size: 200,000 Work Items

Programmer decides to use a Local Work Group Size of 100 Work Items
→ Number of Work Groups: 2,000 Work Groups

One Work Item requires 10 registers and 20 byte of Shared Memory; a SM has 16 KB of Shared Memory and 16,384 registers
→ Number of Work Items per SM:
  Max(16,384/10, 16KB/20B) = 819 Work Items
→ Number of Work Groups per SM:
  819/100 = 8 Work Groups per SM

Even if 7 Work Groups are waiting for memory, 1 can be executed.
Warp Execution Example

Each of the Work Groups contains 100 Work Items; the Warp Size (native execution size of a SM) is 32

→ Number of Threads Executed in parallel: 32 Threads
→ Number of „Rounds“ to execute a Work Group: 100/32 = 4
→ Threads running in the first 3 rounds: 32 Threads
→ Threads running in the last round: 100−32∗4=4 Threads

If one of the threads accesses memory: whole warp stalls
If one of the threads follows a differing execution path: it is executed in an additional separate round
## OpenCL Platforms

<table>
<thead>
<tr>
<th>Platform</th>
<th>Compatibility</th>
<th>Linux/Windows/OS Support</th>
<th>GPU Support</th>
<th>Additional Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AMD APP SDK</strong> (supports OpenCL CPU and accelerated processing unit Devices)**</td>
<td>X86 + SSE2 (or higher) compatible CPUs 64-bit &amp; 32-bit;[22] Linux 2.6 PC, Windows Vista/7 PC</td>
<td><strong>AMD Fusion</strong> E-350, E-240, C-50, C-30 with HD 6310/HD 6250</td>
<td><strong>AMD Radeon/Mobility HD 6800, HD 5x00 series GPU, IGPU HD 6310/HD 6250</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Intel SDK for OpenCL Applications 2013</strong>[78] (supports Intel Core processors and Intel HD Graphics 4000/2500)</td>
<td><strong>Intel</strong> CPUs with SSE 4.1, SSE 4.2 or AVX support, [79][80] Microsoft Windows, Linux</td>
<td>**Intel Core i7, i5, i3; 2nd Generation Intel Core i7/5/3, 3rd Generation Intel Core Processors with Intel HD Graphics 4000/2500</td>
<td><strong>Intel Core 2 Solo, Duo Quad, Extreme</strong></td>
<td></td>
</tr>
<tr>
<td><strong>IBM Servers with OpenCL Development Kit for Linux on Power running on Power VSX</strong>[81][82]</td>
<td>IBM Power 755 (PERCS), 750</td>
<td>IBM BladeCenter PS70x Express</td>
<td>IBM BladeCenter JS2x, JS43</td>
<td>IBM BladeCenter QS22</td>
</tr>
<tr>
<td><strong>IBM OpenCL Common Runtime (OCR)</strong>[83]</td>
<td>X86 + SSE2 (or higher) compatible CPUs 64-bit &amp; 32-bit;[84] Linux 2.6 PC</td>
<td><strong>AMD Fusion, Nvidia Ion and Intel Core i7, i5, i3; 2nd Generation Intel Core i7/5/3</strong></td>
<td><strong>AMD Radeon, Nvidia GeForce and Intel Core 2 Solo, Duo, Quad, Extreme</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Nvidia OpenCL Driver and Tools</strong>[85]</td>
<td>Nvidia Tesla C/D/S</td>
<td>Nvidia GeForce GTS/GT/GTX</td>
<td>Nvidia Ion</td>
<td>Nvidia Quadro FX/NVX/Plex</td>
</tr>
</tbody>
</table>

[22] Linux 2.6 PC, Windows Vista/7 PC

[78] Intel Core processors and Intel HD Graphics 4000/2500

[79] Intel HD Graphics 4000/2500

[80] AMD Radeon, Nvidia GeForce and Intel Core 2 Solo, Duo, Quad, Extreme

[81] IBM Power 755 (PERCS), 750

[82] IBM BladeCenter PS70x Express

[83] IBM BladeCenter JS2x, JS43

[84] Nvidia GeForce GTS/GT/GTX

[85] Nvidia Ion
## Compute Capability by version

<table>
<thead>
<tr>
<th>Feature</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
<th>3.0</th>
<th>Since 3.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double precision floating point operations</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Caches</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Max # concurrent kernels</td>
<td>1</td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td></td>
<td>Dynamic Parallelism</td>
</tr>
<tr>
<td>Max # threads per block</td>
<td>512</td>
<td></td>
<td></td>
<td></td>
<td>1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max # Warps per MP</td>
<td>24</td>
<td>32</td>
<td></td>
<td>48</td>
<td></td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>Max # Threads per MP</td>
<td>768</td>
<td>1024</td>
<td></td>
<td>1536</td>
<td></td>
<td>2048</td>
<td></td>
</tr>
<tr>
<td>Register count (32 bit)</td>
<td>8192</td>
<td>16384</td>
<td></td>
<td>32768</td>
<td></td>
<td>65536</td>
<td></td>
</tr>
<tr>
<td>Max shared mem per MP</td>
<td>16KB</td>
<td></td>
<td></td>
<td>16/48KB</td>
<td></td>
<td>48-112KB</td>
<td></td>
</tr>
<tr>
<td># shared memory banks</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>32</td>
<td></td>
</tr>
</tbody>
</table>

**Plus:** varying amounts of cores, global memory sizes, bandwidth, clock speeds (core, memory), bus width, memory access penalties ...

Programmierung paralleler und verteilter Systeme | GPU Computing | FF2015
big performance gains for small problem sizes

* less is better
small/moderate performance gains for large problem sizes → further optimizations needed

* less is better
## Best Practices for Performance Tuning

<table>
<thead>
<tr>
<th>Category</th>
<th>Best Practices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm Design</td>
<td>Asynchronous, Recompute, Simple</td>
</tr>
<tr>
<td>Memory Transfer</td>
<td>Chaining, Overlap Transfer &amp; Compute</td>
</tr>
<tr>
<td>Control Flow</td>
<td>Divergent Branching, Predication</td>
</tr>
<tr>
<td>Memory Types</td>
<td>Local Memory as Cache, rare resource</td>
</tr>
<tr>
<td>Memory Access</td>
<td>Coalescing, Bank Conflicts</td>
</tr>
<tr>
<td>Sizing</td>
<td>Execution Size, Evaluation</td>
</tr>
<tr>
<td>Instructions</td>
<td>Shifting, Fused Multiply, Vector Types</td>
</tr>
<tr>
<td>Precision</td>
<td>Native Math Functions, Build Options</td>
</tr>
</tbody>
</table>
Divergent Branching and Predication

Divergent Branching
- Flow control instruction (if, switch, do, for, while) can result in different execution paths
  - Data parallel execution → varying execution paths will be serialized
  - Threads converge back to same execution path after completion

Branch Predication
- Instructions are associated with a per-thread condition code (predicate)
  - All instructions are scheduled for execution
  - Predicate true: executed normally
  - Predicate false: do not write results, do not evaluate addresses, do not read operands
- Compiler may use branch predication for if or switch statements
- Unroll loops yourself (or use #pragma unroll for NVIDIA)
Coalesced Memory Accesses

**Simple Access Pattern**
- Can be fetched in a single 64-byte transaction (red rectangle)
- Could also be permuted *

**Sequential but Misaligned Access**
- Fall into single 128-byte segment:
  - single 128-byte transaction, else: 64-byte transaction + 32-byte transaction *

**Strided Accesses**
- Depending on stride from 1 (here) up to 16 transactions *
* 16 transactions with compute capability 1.1
Use Caching: Local, Texture, Constant

**Local Memory**
- Memory latency roughly 100x lower than global memory latency
- Small, no coalescing problems, prone to memory bank conflicts

**Texture Memory**
- 2-dimensionally cached, read-only
- Can be used to avoid uncoalesced loads from global memory
- Used with the image data type

**Constant Memory**
- Lineary cached, read-only, 64 KB
- as fast as reading from a register for the same address
- Can be used for big lists of input arguments
Memory Bank Conflicts

- Access to (Shared) Memory is implemented via hardware memory banks
- If a thread accesses a memory address this is handled by the responsible memory bank
- Simple Access Patterns like this one are fetched in a single transaction
Memory Bank Conflicts

Permuted Memory Access (left)
- Still one transaction on cards with compute capability $\geq 1.2$; otherwise 16 transactions are required.

Strided Memory Access (right)
- Still one transaction on cards with compute capability $\geq 1.2$; otherwise 16 transactions are required.
Bank conflicts

- Left figure: 2 bank conflicts → resulting bandwidth is $\frac{1}{2}$ of the original bandwidth

- Right figure: 8 bank conflicts → resulting bandwidth is $\frac{1}{8}$ of the original bandwidth
Sizing: What is the right execution layout?

- Local work item count should be a multiple of native execution size (NVIDIA 32, AMD 64), but not too big.
- Number of work groups should be multiple of the number of multiprocessors (hundreds or thousands of work groups).
- Can be configured in 1-, 2- or 3-dimensional layout: consider access patterns and caching.
- Balance between latency hiding and resource utilization.
- Experimenting is required!
Instructions and Precision

- Single precision floats provide best performance
- Use shift operations to avoid expensive division and modulo calculations
- Special compiler flags
- AMD has native vector type implementation; NVIDIA is scalar
- Use the native math library whenever speed trumps precision

<table>
<thead>
<tr>
<th>Functions</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>single-precision floating-point add, multiply, and multiply-add</td>
<td>8 operations per clock cycle</td>
</tr>
<tr>
<td>single-precision reciprocal, reciprocal square root, and native_logf(x)</td>
<td>2 operations per clock cycle</td>
</tr>
<tr>
<td>native_sin, native_cos, native_exp</td>
<td>1 operation per clock cycle</td>
</tr>
</tbody>
</table>
Further Readings

http://www.dcl.hpi.uni-potsdam.de/research/gpureadings/

- [7] Rob Farber. CUDA, Supercomputing for the Masses. Dr. Dobb’s
- [8] NVIDIA. OpenCL Programming for the CUDA Architecture
- [9] Ryan Smith, NVIDIA’s GeForce GTX 480 and GTX 470: 6 Months Late, Was It Worth the Wait?
- [10] Stephen Jones. Introduction to Dynamic Parallelism