Parallel Programming Concepts

GPU Computing with OpenCL

Frank Feinbube

Operating Systems and Middleware
Prof. Dr. Andreas Polze
## Parallel Programming Models

<table>
<thead>
<tr>
<th>Programming Model</th>
<th>Programming Language / Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Tasking</td>
<td>pthreads, <strong>OPENCL</strong>, Linda, Ada, Cilk, PGAS, ...</td>
</tr>
<tr>
<td>Message Passing</td>
<td>MPI, PVM, CSP channels, actors ...</td>
</tr>
<tr>
<td>Implicit Parallelism</td>
<td>Map/Reduce, PLINQ, HPF, Lisp, Fortress ...</td>
</tr>
<tr>
<td>Mixed</td>
<td>OpenMP &amp; Co., Scala, Erlang, X10, Clojure...</td>
</tr>
</tbody>
</table>
The Parallel Programming Problem

Flexible

Parallel Application

Match ?

Execution Environment

Fixed
CPU vs. GPU Architecture
GF100

32,768 Registers

32 Scalar Processors

4 Special Function Units (Math)

Configurable Shared Memory / Cache (16KB + 48KB)
Execution Environment

- **Shared nothing**
  - Pfister: "distributed memory"
  - Foster: "multicomputer"
  - Tanenbaum: "private memory"

- **Shared memory**
  - Pfister: "shared memory"
  - Foster: "multiprocessor"
  - Tanenbaum: "shared memory"
Data Parallelism and Task Parallelism

Data Parallelism

Input Data
Parallel Processing
Result Data

Task Parallelism

Aggregation Task

Aggregation Task
### Execution Environment Examples

<table>
<thead>
<tr>
<th>Execution Environment</th>
<th>Data Parallel / SIMD</th>
<th>Task Parallel / MIMD</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Shared Memory (SM)</strong></td>
<td>SM-SIMD</td>
<td>SM-MIMD ManyCore/SMP system</td>
</tr>
<tr>
<td></td>
<td>GPU, Cell, SSE, AltiVec Vector processor</td>
<td>...</td>
</tr>
<tr>
<td><strong>Shared Nothing / Distributed Memory (DM)</strong></td>
<td>DM-SIMD</td>
<td>DM-MIMD</td>
</tr>
<tr>
<td></td>
<td>processor-array systems Hadoop</td>
<td>cluster systems MPP systems</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Note: Task-parallel execution environments are easily also usable as data-parallel execution environment, but not optimized for it.
The Parallel Programming Problem

Parallel Application

Match ?

Execution Environment (SM/DM + SIMD/MIMD)

Flexible

Fixed
Wide Variety of Application Domains


http://www.nvidia.com/object/tesla_testimonials.html
## History of GPU Computing

<table>
<thead>
<tr>
<th>Category</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed Function Graphic Pipelines</strong></td>
<td>1980s-1990s; configurable, not programmable; first APIs (DirectX, OpenGL); Vertex Processing</td>
</tr>
<tr>
<td><strong>Programmable Real-Time Graphics</strong></td>
<td>Since 2001: APIs for Vertex Shading, Pixel Shading and access to texture; DirectX9</td>
</tr>
<tr>
<td><strong>Unified Graphics and Computing Processors</strong></td>
<td>2006: NVIDIA's G80; unified processors arrays; three programmable shading stages; DirectX10</td>
</tr>
<tr>
<td><strong>General Purpose GPU (GPGPU)</strong></td>
<td>Compute problem as native graphic operations; algorithms as shaders; data in textures</td>
</tr>
<tr>
<td><strong>GPU Computing</strong></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
  - Was tired of recoding for many core, GPUs. Pushed vendors to standardize.

- **Intel**

- **Apple**

Khronos Compute Group formed

Wrote a draft straw man API

---

**OpenCL**

[1] AMD

[2] ATI

[3] NVIDIA

[4] Intel

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 and NVIDIA GPUs, x86 CPUs

- 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: focus on 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
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.
OpenCL Execution Model

An example of an NDRange index space showing work-items, their global IDs and their mapping onto the pair of work-group and local IDs.

OpenCL Execution Model

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. [5]
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 (__global const float *a,
                       __global const float *b,
                       __global float *c)
{
    int gid = get_global_id(0);
    c[gid] = a[gid] + b[gid];
}
```

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

```c
// create the OpenCL context on a GPU device
cl_context = clCreateContextFromType(0,
    CL_DEVICE_TYPE_GPU, 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_float)*n, &cb,
    NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
    CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, &cb,
    NULL);
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(cl_float)*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_float), dst, 0, NULL, NULL);
```

`Define platform and queues`

```
defines[0], 0, NULL);
```

`Define Memory objects`

```
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, sizeof(cl_float)*n, SrcB,
memobjs[2] = clCreateBuffer(context, CL_MEM_WRITE_ONLY, ...
```

`Create the program`

```
Create and setup kernel
```

```
// 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));
```

`Build the program`

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

`Execute the kernel`

```
// read results on the host
```

```
true, 0,
```

“standard” overhead for an OpenCL program
Live Demo

OpenCL “Hello Device”

OpenCL “Sudoku Validator”
**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:**
- NVIDIA’s Parallel Nsight for Microsoft Visual Studio
- AMD’s ATI Stream Profiler
- AMD’s Stream KernelAnalyzer: displays GPU assembler code, detects execution bottlenecks
- gDEBugger (platform-independent)

Big knowledge bases with tutorials, examples, articles, show cases, and developer forums
## 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>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>double precision floating point operations</td>
<td>No</td>
<td></td>
<td></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>caches</td>
<td>No</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>
</tr>
<tr>
<td>max # threads per block</td>
<td>512</td>
<td>1024</td>
<td></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>
</tr>
<tr>
<td>max # Threads per MP</td>
<td>768</td>
<td>1024</td>
<td></td>
<td>1536</td>
<td></td>
</tr>
<tr>
<td>register count (32 bit)</td>
<td>8192</td>
<td>16384</td>
<td></td>
<td>32768</td>
<td></td>
</tr>
<tr>
<td>max shared mem per MP</td>
<td>16KB</td>
<td></td>
<td></td>
<td>48KB</td>
<td></td>
</tr>
<tr>
<td># shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

big performance gains for small problem sizes

Intel E8500 CPU
AMD R800 GPU
NVIDIA GT200 GPU

* 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>Tips</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>
Parallel Application

Coarse-grained vs. Fine-grained

Multi-Tasking
Message Passing
Implicit Parallelism
Mixed stuff

<table>
<thead>
<tr>
<th></th>
<th>Scaleup</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM-MIMD</td>
<td>(Inter)</td>
<td>Intra</td>
</tr>
<tr>
<td>DM-MIMD</td>
<td>Inter</td>
<td>(Intra)</td>
</tr>
</tbody>
</table>

Intra-request parallelism for response time
-> speedup

Inter-request parallelism for throughput
-> scaleup
Further Readings

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

- [7] Rob Farber, 2008. CUDA, Supercomputing for the Masses. Dr. Dobb’s
- [9] Ryan Smith, NVIDIA’s GeForce GTX 480 and GTX 470: 6 Months Late, Was It Worth the Wait?