Kernel Acceleration on Heterogeneous Many-Core Architectures

Giovanni Agosta, Alessandro Barenghi, Gerardo Pelosi and Michele Scandale
Politecnico di Milano, Italy

28/01/2014
Introduction
Moore’s Wall

Issue
- Clock speed are not improving at the same rate they did in the last 40 years!
- Transistor density still improving according to Moore’s Law
- But increases in register bank, cache size, pipeline depth are hitting the point of diminishing returns:
  - Cache size increases are useful when cache hit rate is low
  - When the hit rate becomes very high, increasing the cache size will yield minimal performance benefits

Solution
Replace single, complex superscalar processors with numerous but smaller and simpler processing units!
Evolution of number of cores

- Rapid growth in the number of computing cores per chip
- High-end embedded CPUs evolved from 1 to 8 designs (e.g., ARM big.LITTLE) in the last 3 years!
- Specialized architectures, such as Graphics Processing Units, have 100s of cores: many-core architectures
### Notable many-core architectures

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Desktop GPGPUs</strong></td>
<td>NVIDIA GT200, Fermi and Kepler, AMD R700 and R800</td>
</tr>
<tr>
<td><strong>Embedded GPGPUs</strong></td>
<td>ImgTech PowerVR, NVIDIA Tegra</td>
</tr>
<tr>
<td><strong>Non-GPU Coprocessors</strong></td>
<td>IBM CellBE, Intel Xeon Phi, Adapteva Epiphany</td>
</tr>
<tr>
<td><strong>Many-core Standalone Systems</strong></td>
<td>Intel SCC</td>
</tr>
</tbody>
</table>
State of the art

- GPGPUs are dominating the many-core scene
- Non-GPU accelerators have found application in specialized domains
- Many-core architectures are not well suited for control-intensive applications
- Emerging paradigm: multi-core host architecture plus (one or more) many-core accelerator device(s)
nVidia GPGPU Architecture

GT200 Multiprocessor
- Thread Pool (768 threads)
- MT-Issue
- 16,384 Registers
- 32 kibiB Shared Memory

Fermi Multiprocessor
- Thread Pool (768 threads)
- MT-Issue
- 32,768 Registers
- 64 kibiB Shared Memory

Kepler Multiprocessor
- Instruction Cache
- Thread Pool
- 65,536 Registers
- 64 kibiB Shared Memory

- 4x MT-Issue
- SFU

Agosta, Barenghi, Pelosi, Scandale
Many-Core Acceleration
Non-GPU accelerators are expected to be more versatile, and may become a dominant paradigm in the future.

The classification above might be overcome through *convergence*.

Heterogeneity will still play a role.
Introduction

How to Program Parallel Architectures?

Shared Memory

- OpenMP directives
- Threads
- Cilk or other specialized languages

Unfortunately, shared memory does not scale well

Distributed Memory

- MPI (message passing interface), works well for distributed memory, but is quite complex to handle
- Partitioned Global Address Space (PGAS) languages, easier to use but require a virtual shared memory
Introduction
How to Program Parallel Architectures?

Heterogeneous Architectures
- Asymmetric systems where host processors has different capabilities from accelerator processors
- Typical of many-core architectures
- Emerging paradigm: multi-core host architecture plus (one or more) many-core accelerator device(s)

Programming Languages and APIs
- OpenCL, currently the most popular
- OpenACC, a newer proposal
- Heterogeneous variants of OpenMP
<table>
<thead>
<tr>
<th>GPGPUs have some specificities w.r.t. other many-cores</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Control flow divergence</strong></td>
</tr>
<tr>
<td><strong>Hardware design of GPGPUs</strong></td>
</tr>
<tr>
<td><strong>Local memory availability</strong></td>
</tr>
</tbody>
</table>
General Purpose GPUs
Types of parallelism in GPGPUs

Three main kinds of parallelism

**Thread-level parallelism**  Not very suited for GPGPUs, due to impact of control flow divergence, but can be managed at a coarser grain

**Loop-level parallelism**  An excellent fit for vector processors, SIMD processors and GPGPUs: fixed control flow identical for all iterations

**Instruction-level parallelism**  Mostly unsuitable for GPGPUs due to the need to execute different instructions simultaneously
General Purpose GPUs
nVidia GPU Architectures

Clustered many-core architecture

- A large set of *Stream Processors* grouped in *stream multiprocessors*
- Each *Stream Processors* is a simple single-issue computing units with an integer ALU and a FPU
- Also, dedicated units for graphic rendering workloads
- The stream multiprocessors are composed by a group of stream processors sharing one or more instruction issue units, a common register file, and an addressable local cache
- A memory controller is employed to access the GPU card global memory
Introduction
Modern GPGPUs
The OpenCL Programming model

General Purpose GPUs
nVidia GPGPU Architectures

GT200, Fermi and Kepler

GT200 Multiprocessor
- Thread Pool (768 threads)
- MT-Issue
- 16,384 Registers
- 32 kiB Shared Memory

Fermi Multiprocessor
- Thread Pool (768 threads)
- MT-Issue
- MT-Issue
- 32,768 Registers
- 64 kiB Shared Memory

Kepler Multiprocessor
- Instruction Cache
- Thread Pool
- 4x MT-Issue
- 65,536 Registers
- 64 kiB Shared Memory

Agosta, Barenghi, Pelosi, Scandale
Many-Core Acceleration
General Purpose GPUs

GT200 Architecture

- 8 stream processors per stream multiprocessor, running at twice the clock of the GPU
- The single (shared) issue unit runs at half of the clock frequency of the GPU
- Each instruction issued is effectively executed 32 times on different data (a warp)
- Control flow divergence is handled by computing all branches, and committing only the actually taken ones
- Issued warp is chosen among 24 different ones belonging to at most 3 different work-groups
- 16 load-store units are present in the stream multiprocessor
General Purpose GPUs
Fermi Architecture

- 32 stream processors per stream multiprocessor
- Two issue units, running at the full GPU clock speed
- 64 instructions coming from two different warps are executed at each clock cycle
- Register file and local shared addressable cache are doubled in size with respect to the GT200
- Additional L2 non addressable data cache of 768 kiB, shared by all the stream multiprocessors
General Purpose GPUs
Kepler Architecture

- Twice as many registers as Fermi
- 192 stream processors per stream multiprocessor is raised to 192
- 192 instructions executed per clock cycle
- 32 load-store units per stream multiprocessor
- Four issue units per stream multiprocessor, larger instruction buffer (64 warps), and an instruction cache
- L2 cache increased to 1536 kiB wide, with six separate memory controllers
- Better handling of irregular memory access patterns
- Dynamic parallelism support
General Purpose GPUs
AMD GPU architectures

Structure of the stream processors and stream multiprocessors in the R700 and R800 architectures

R700 Stream Processor
A-Unit
A-Unit
A-Unit
A-Unit
T-Unit
Branch Unit
16,384 Registers

Barts SIMD Core
Thread Sequencer
Thread Sequencer
SP SP
SP SP

R700 Full Chip Architecture
Chip Level Thread Dispatcher
Constant Cache
Instruction Cache
Intra-SIMD Shared Mem
SIMD Core
SIMD Core
SIMD Core
SIMD Core
SIMD Core
SIMD Core
SIMD Core
SIMD Core

R800 Stream Processor
T-Unit
T-Unit
T-Unit
T-Unit
Branch Unit
16,384 Registers

Barts SIMD Core
Texture Unit
Texture Unit
SP SP
SP SP

SIMD Core
SIMD Core
SIMD Core
SIMD Core
SIMD Core
SIMD Core
SIMD Core
SIMD Core

Memory Controller
128 kB L2 Cache
128 kB L2 Cache
128 kB L2 Cache
128 kB L2 Cache

128 kiB
Texture Cache
32 kiB
Shared Memory

Agosta, Barenghi, Pelosi, Scandale
Many-Core Acceleration
R700

- More complex stream processor handling fine grained parallelism to reduce control flow divergence penalties
- A single AMD stream processor is a VLIW unit endowed with a branch handling lane
- R700 has VLIW processors with four ALUs and one unit able to compute transcendental functions
- R700 has 16384-registers-wide register file
- SIMD cores (stream multiprocessors) of 16 stream processors
- Single instruction issue unit and 8 kiB local shared memory
- Efficient barrier synchronization
R800

- R800 has four general purpose units instead of the four ALU and one general purpose unit
- R700 has 128 bit registers, rather than 32 bit ones
- 32 kiB local shared memory
OpenCL (Open Computing Language)

- Open standard for development of parallel applications on heterogeneous multi-core architectures
- Handles and can combine multiple platforms (GPUs, CPUs, DSPs)
- Widely accepted in and supported by industry

Structure

- **OpenCL-C** a subset of C99 with appropriate language extensions
- **OpenCL API** which allows programs to be split into a “host part” and a “compute device part”
OpenCL Programming Model

Host Code

- Runs on a general purpose (multi-) processor, is in charge of executing the control-intensive code
- Uses the OpenCL API to query and select compute devices
- Offloads compute-intensive code (*kernels*) on compute devices
- Work-queues are used to manage kernel offloading
### Kernel execution

- Kernel execution handled as a double nested loop
- Inner loop iterations execute kernel code in independent execution elements (*work-item*)
- Outer loop iterations gather work-items in independents sets (*work-group*)
- The computation domain of the kernel (data placement) can be thought as an \(N\)-dimensional domain
- Work-item have unique identifier composed by \(N\) unsigned integer values
- Work-groups are uniquely identified through a set of unsigned integer values in \([0, N - 1]\)
Data Parallelism

- Explicitly parallel function invocation (*kernel*) executed by a user-specified number of work-items set in an abstract $N$-dimensional space
- Kernel execution is asynchronously started through a `clEnqueueNDRangeKernel` call

Task Parallelism

- Multiple kernels can be enqueued for execution
- They may be run in parallel by the underlying hardware of the compute device
- Events can be used to provide a dependency relation among the kernels
Work-groups and work-items

- The OpenCL programming model expresses concurrency through the concepts of work-group and work-item.
- A work-group captures the notion of a group of concurrent work-items.
- Work-groups are required to be computed independently.
- OpenCL-C synchronization primitives act only among work-items in a work-group.
- Kernel calls must specify the number of work-groups and work-items.
The work-groups and work-items can be laid out in a multi-dimensional grid

- **work_dim**: Number $N$ of dimensions used to describe the work-item grid.
- **global_work_offset**: Start offset for each dimension (so that the grid origin of the axes may be different from zero).
- **global_work_size**: Total number of work-items, for each dimension.
- **local_work_size**: Number of work-items in each work-group, for each dimension.

Note that OpenCL has no limits on $N$, platform introspection is used to detect platform-specific constraints.
Platform constraints on work-space dimensions

The following constants can be passed to `clGetDeviceInfo` to obtain the constraints for the aforementioned parameters:

- **CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS**: Maximum number of dimensions in the work-item grid.
- **CL_DEVICE_MAX_WORK_GROUP_SIZE**: Maximum number of work-items in a work-group.
- **CL_DEVICE_MAX_WORK_ITEM_SIZES**: Maximum number of work-items in each dimension of the work-group.
OpenCL Memory Model

Overview

- Explicit memory hierarchy model
- Memory model distributed between the host and the compute device
- Multiple address spaces
- The device *global memory* is shared among all work-items regardless of the work-group
- The host is allowed to read-from and write-to the device memory space only through the OpenCL API
OpenCL Memory Model
Overview

Host
- Host Memory

Compute Device
- Global Memory
- Constant Memory
- Local Memory
  - Work Item: Private Memory
  - Work Item: Private Memory
  - Work Item: Private Memory
  - Work Item: Private Memory

Agosta, Barenghi, Pelosi, Scandale
Many-Core Acceleration
OpenCL Memory Model
Memory Address Spaces

Local and Global Memory

- *Local memory* associated with each work-group
- Mapped to on-chip memory for faster access
- Intra-work-group shared-memory data transfer
- Synchronized through an explicit barrier
- Work-items belonging to different work-groups must communicate through *global memory*

Constant and Private Memory

- *Constant memory* shared among all work-items
- *Private memory*, non-shared

Qualifiers: __global, __local, __constant and __private
### OpenCL Memory Model

#### OpenCL Memory Regions

<table>
<thead>
<tr>
<th>Allocation and access capabilities of both host and compute device for the four OpenCL memory address spaces</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host allocation</strong></td>
</tr>
<tr>
<td>Dynamic</td>
</tr>
<tr>
<td>None</td>
</tr>
<tr>
<td>Dynamic</td>
</tr>
<tr>
<td>None</td>
</tr>
</tbody>
</table>

Dynamic memory allocation and recursion not available on the device.
A simple program which computes the square of the first \( n \) natural numbers (where \( n \) is an argument of the program)

Each square will be computed in a different work-item, collecting them in work-groups of 8

The host employs an OpenCL-c kernel that computes the square of each element of an array of integers

The host code uses the C++ bindings for the OpenCL API (OpenCL 1.1 or more)

Note that work-group formation may be driven by architectural considerations

How many work-items can be executed at the same time?
OpenCL Programming

Host-side code

Directives (including C++ standard header files)

```cpp
#define __CL_ENABLE_EXCEPTIONS 1
#include <vector>
#include <iostream>
#include <sstream>
#include <string>
#include <CL/cl.hpp>
using namespace cl;
```

- `__CL_ENABLE_EXCEPTIONS` selects C++ exceptions rather than C error handling
- `CL/cl.hpp` provides OpenCL API C++ bindings in `namespace cl`
The OpenCL kernel

```c
static const std::string source = "\nkernel void square(global int *output,\nglobal int *input){{\n    unsigned int i=get_global_id(0);\n    output[i]=input[i]*input[i];\n}}";
```

- Included as a constant string in the host program
- Three essential elements of any OpenCL-C kernel: the `kernel` keyword, the address spaces, and the work-item identification built-in function `get_global_id`.
The kernel (or __kernel) keyword

- Introduces all entry points in an OpenCL-C program
- Only kernel functions can be invoked from the host
- Non-kernel functions can be defined in OpenCL-C code
- The parameters are arrays in global memory

The get_global_id builtin function

- Maps every work-item to an index in the work-item space
- Parameter: the dimension index (multidimensional work-space)
- Here, the kernel code expects the work-item space to be monodimensional
Argument parsing: get $n$ and store it in $size$

```cpp
int main(int argc, char *argv[]) {
    unsigned int size;
    try {
        std::istringstream arg(argv[1]);
        arg >> size;
    } catch (...) {
        std::cout << "Missing_or_incorrect_argument";
        std::cout << std::endl;
        return 1;
    }
}
```

Note that it is a standard C++ program
Data structure initialization

```cpp
std::vector<cl_int> array_in(size);
std::vector<cl_int> array_out(size);
for(int i=0; i<size; i++) array_in[i]=i;
for(auto &n : array_in) std::cout << n << "\n";
std::cout << std::endl;
```
OpenCL computing platform setup

```cpp
try {
    std::vector<Platform> platforms;
    std::vector<Device> devices;
    Platform::get(&platforms);
    platforms[0].getDevices(CL_DEVICE_TYPE_CPU, &devices);
    Context cxt(devices);
    CommandQueue cmdQ(cxt, devices[0], 0);
}
```

- Boilerplate code using the introspection capabilities of the OpenCL runtime
### OpenCL computing platform setup

- `Platform::get` yields all the available OpenCL platforms (from different vendors)
- Each platform can have multiple devices (e.g., two GPGPUs from the same vendor)
- Create a `Context` and a `CommandQueue` for the selected device
- The `Context` provides all the necessary information for building OpenCL-C kernels
- The `CommandQueue` will be used to actually interact with the devices
Memory buffer allocation

```c
const int in_flags = CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR;
const int out_flags = CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR;
Buffer in(cxt, in_flags, size * sizeof(cl_int), &array_in.at(0));
Buffer out(cxt, out_flags, size * sizeof(cl_int), &array_out.at(0));
```

Memory buffers are used for the kernel execution and for host-device data exchange.
OpenCL Programming

Host-side code

Memory buffer allocation

- The \texttt{in} buffer will be initialized with a copy of the data in \texttt{array\_in}: \texttt{CL\_MEM\_COPY\_HOST\_PTR}
- The \texttt{in} buffer will be read only on the device side: \texttt{CL\_MEM\_READ\_ONLY}
- The \texttt{out} buffer will be mapped in the host memory, since the data produced by the kernel will need to be copied back to the host: \texttt{CL\_MEM\_USE\_HOST\_PTR}
OpenCL Programming

Building the OpenCL-C program

```c
Program program(cxt, source, true);
Kernel kernel(program, "square");
```

- The last parameter of the `Program` constructor specifies that the program must be compiled and linked.
- The `Kernel` constructor selects the entry point by name.
OpenCL Programming

Host-side code

Setup correspondence between kernel parameters (formal vs actual)

```cpp
kernel.setArg<Buffer>(0, out);
kernl.setArg<Buffer>(1, in);
```

- Kernel parameters are identified positionally rather than by their own name on the host side.
Work-space geometry and kernel invocation

```c
NDRange global_range(size);
NDRange local_range(8);
cmdQ.enqueueNDRangeKernel(kernel, NullRange,
                           global_range,
                           local_range);
```

- The two `NDRange` variables indicate the work-item space and the work-group size.
- The second parameter of the `enqueueNDRangeKernel` call specifies that the origin of the work-item space is set at 0.
The execution of OpenCL kernels is per se asynchronous

- **finish** is a blocking method for explicit waiting
- **finish** returns once all commands in `cmdQ` are completed
- Errors are signaled through raising a `cl::Error` exception
Conclusion: printing out the results

```cpp
for(auto &n : array_out) std::cout << n << "\n";
std::cout << std::endl;
return 0;
}
```
For simple kernels the setup code is much larger than the kernel code.

The boilerplate code needs to handle the heterogeneity of the machine (i.e., bridge the host-device divide, through the `Buffer` setup).

It also needs to manage Just-In-Time compilation (here through the `Program` constructor).

Possibly availability of multiple OpenCL runtimes and devices must be handled too.

In real applications, it is possible to select the best platform for a given kernel, and to provide specialized kernel implementations for each device and/or platform.