Programming languages for heterogeneous systems

New opportunities and challenges

Codeplay CEO
Codeplay: GPU Compiler Experts

Been producing GPU compilers since 2002, from work by founders starting 1999

- Developing whole GPU compilers
- Working on customer’s GPU compilers (LLVM and proprietary tech)
- Optimization, bug-fixing, new languages, testing
- Infrastructure for testing and tracking perf
- Collaborative R&D
- Academic collaborations
- Creating test-suites
- Helping define new standards
- GPGPU as well as graphics
- Debuggers, IDE integration
- Graphics and games s/w

35 expert development staff, based in Edinburgh, Scotland, UK
It’s all about power consumption
Power consumption over time

Past: Power consumption trending **up**

Present: Power consumption trending **down**

- PS
- Xbox
- Nintendo
- x86
- Amiga
CPU Clock Frequency Over Time

Past: CPU frequency trends up

Present: CPU frequency trends flat
We are in a new era:
heterogeneous computing

This impacts language design. There are new problems we need to solve
The growth of the GPU in mobile: Apple’s A4-A8

Source: Chipworks
The key drivers of heterogeneous computing: data movement cost

- Cost of data movement is much higher than computation cost
- GPUs control data movement distances carefully
- Preserve locality explicitly instead of caching

Source: NVIDIA: Bill Dally’s presentation at SC10
The key drivers of heterogeneous computing: throughput vs latency

- Long pipelines allow very high clock frequencies
- Large number of cores allows very high throughput
- Each delivers best efficiency for different kinds of algorithm
Designing easy-to-use software systems

How do we handle heterogeneous processing in software?
Horizontal vs vertical approaches

**Vertical approach:**
Everything proprietary

**Horizontal approach:**
standard layers

User’s software

- Programming model A
  - Core A
- Programming model B
  - Core B
- Programming model C
  - Core C

User’s software

- Libraries
- Language X
- Standard binary format
- Driver A
  - Core A
- Driver B
  - Core B
- Driver C
  - Core C

Industry moving in this direction
Direction industry is moving

• Standardizing binary formats for cores
  – OpenCL SPIR, HSAIL, plus others coming

• Standardizing languages and runtimes:
  – SYCL, C++ AMP, OpenMP, Parallel STL, plus others coming

• Next steps:
  – Standardizing tools capabilities, like debugging

• Benefits are: easier programming models from more suppliers for more cores: more innovation
OpenCL SPIR

• Available now from multiple vendors
• A standard binary format extension to OpenCL
• Works with the OpenCL API, but lets you write your own compilers on top
• Based on top of Clang/LLVM
  – very good, maintained, open-source compiler tool-set
HSA

- Lower level than OpenCL and SPIR
- Provisional specifications out now
  - Standard still under development
- Close to heterogeneous hardware level
  - Developers can optimize for very low-latency communication
What do we need in a programming model?
Things to abstract away

1. Throughput vs low-latency processor cores
2. Different forms of memory and data sharing
3. The different compilers and instruction sets
4. Performance optimization differences
Throughput vs low-latency cores
Throughput vs low-latency

**Throughput**
- Code broken up into kernels or tasks
- Usually, data-parallel kernels that execute loops in parallel
- Build tasks into task-graphs that can be scheduled efficiently
- Data structures should be arrays where possible
- Data movement is part of the task graph

**Low-latency CPUs**
- Can respond rapidly to inputs and conditions
- Code will be complex threads
- Often the bottleneck when targeting performance – so minimize code for these cores
- Can handle complex data structures and branching
- Data movement is via caches and is on-demand
The approach in SYCL

- Data-parallel loops are written has C++ `parallel_for` function
  - This is the standard approach for C++
- SYCL has 2 levels of `parallel_for` for the different levels of data parallelism in GPUs
- Separate data storage (`buffers`) from data access (`accessors`)
- Kernels are built up into a task graph and not executed immediately
Different forms of memory & data sharing
Data movement in software

- Data movement is hidden in most current programming languages:

  a = b + c;  3 data move ops, but what are they?

  a = b [i];  2 data movement ops, only 1 explicit
Handling data movement cost

• There is little agreement on how to reduce data movement costs in hardware design
  – We are still working our way around the design space

• This is an *abstraction* problem:
  i. How do we give programmers *control* over high costs?
  ii. How do we let programmers *hide* implementation details?
Managing data movement in hardware

Execution core

Registers

On-chip memory

Per-device DRAM

DRAM – shared across whole system

Distance from core
Impact on the programmer of data movement

Execution core

Execution cores

On-chip memory

Per-device

DRAM – shared across whole system

Sharing, accessibility, addressing, coherency vary between different levels on different devices
Different forms of data access

• What happens when we write:
  \[ a = b[i]; \]

• The contents of \( b[i] \) need to be moved into a register on an execution core

• But, they may have to be moved through multiple levels of cache and/or memory to get there
Impact on the programmer of data movement

- Execution core
- Registers
- On-chip memory
- Registers
- Execution cores
- On-chip memory
- Per-device DRAM
- DRAM - shared across whole system

Distance from core
Impact on the programmer of data movement

In a caching system, the contents of \( b[i] \) are moved on-demand to the execution core's local memory.

In a DMA system, the contents of all of \( b[i] \) must be moved in advance to the execution core's local memory.
# Caching vs DMA

## Caching
- Low-latency: can access any data quickly
- In a virtual memory environment, requires OS and hardware support
- Lower bandwidth: how much? Good question!

## DMA
- High-latency: must pre-move data
- Easy to implement in hardware and software
- Easy to add more cores
- Higher bandwidth: how much? Good question!
Integrating multiple compilers

Some alternative approaches and their impacts
Separate-source vs single-source

```c
void mySort ()
{
    MyDataType myData = getData();
    runKernel ("mySortKernel");
}

void mySortKernel (MyDataType *myData)
{
    // sort myData
}
```
Separate-source vs single-source

Separate-source

- Can modify code on the fly
- Easier to implement
  - Datatypes need to be hand-duplicated and checked
  - Can’t template across device boundaries

Single-source

- Compiled offline
- Can share source between processor cores
- Type-checked
- Can provide a single generic function for host & devices
Compiling separate source

- OpenCL is separate source
Single source: single compiler

- This is the common single-source approach
- Have a single pre-processor and compiler front-end, and multiple compiler back-ends
Single source: shared source approach

- Have a different compiler for host and each device
  - Don’t really need to implement different front-end for each device, but can
- Need to link the different device and host code together
  - Device compiler must extract kernels from source & output glue code
Performance optimization differences
Balance of abstraction

**High level abstraction**
- How do you drill down to low-level performance details?
- Providing consistent performance when the program changes
- Providing performance profiling information to a user they can act on

**Low-level access**
- How do you maintain portability?
- Enabling automatic optimization
- How do you bring old software to new systems?
- How do you design libraries that are usable and customizable by users?
Approaches to performance differences

• Patterns & skeletons: using metaprogramming to build up standard algorithms (e.g. C++ Parallel STL)
  – An advanced user can improve optimization themselves

• Compiler transformations (e.g. vectorization and scalarization)
  – Only the compiler developer can improve optimization
How should we judge a solution?

1. Does it interoperate with other solutions?
2. Can an advanced developer drill-down through abstractions to achieve performance benefits?
3. Can problems (bugs or performance) be solved by following instructions?
4. Will it continue to work with future architectures?
What now?

• Problems we need to solve:
  – What should a standard tool-set for heterogeneous systems look like?
    • Does normal compiler->linker->executable->loader order still make sense?
  – How to debug heterogeneous systems?
  – What should languages look like?
  – How to do high-level language features: garbage-collection, JIT, virtual methods ...
What now?

• What do we need to build?
  – Solid tools that work together
  – Tools that provide full software development capability across multiple architectures in a system: compiling, linking, profiling, debugging
  – Libraries that can be user-customized and work together