ARM

- A company licensing IP to all major semiconductor companies (form of R&D outsourcing)
  - Established in 1990 (spin-out of Acorn Computers)
  - Headquartered in Cambridge, UK, with 2500 employees across 27 offices in 13 countries

- ARM is the most widely used 32-bit CPU architecture
  - Dates back to the mid 1980s (Acorn RISC Machine)
  - Dominant in mobile devices (e.g. on average 3 processors per phone)

- Mali is the most widely licensed GPU architecture
  - Dates back to the early 2000s (developed by Falanx, Norway)
  - Media Processing Division established in 2006 (acquisition of Falanx)
  - Mali products
    - Mali-55 (OpenGL ES 1.1), Mali-200, Mali-400 (OpenGL ES 2.0)
    - Mali-T600 (OpenGL ES 3.0 + OpenCL 1.2)
Exynos 5250: dual A15 + quad T600

Google Nexus 10 ($449): 2560x1600 (WQXGA)
http://www.google.com/nexus/10

Chromebook Series 3 ($249)
http://www.google.com/chromebook

Arndale Board ($249)
http://www.arndaleboard.org

Mont-Blanc Supercomputer (EU-funded 😊)
http://www.montblanc-project.eu
OVERVIEW OF ARCHITECTURES AND PROGRAMMING MODELS
**Introduction**

- Special-purpose HW can outperform general-purpose HW
  - Sometimes, by orders of magnitude
  - Importantly, in terms of energy efficiency as well as raw speed
  - Parallel execution is key (more on this later)
- Non-programmable / somewhat-programmable accelerators
  - ASICs, FPGAs, DSPs, early GPUs
- Programmable accelerators
  - Vector extensions: x86/SSE/AVX, ARM/NEON, PowerPC/VMX
  - Recent GPUs supporting general-purpose computing (GPGPUs)
  - Sony/Toshiba/IBM Cell (Sony PlayStation 3, HPC)
  - ClearSpeed CSX (HPC, embedded)
  - Adapteva Epiphany (HPC, mobile)
  - Intel Xeon Phi (HPC)
Landscape of accelerator programming

- Plethora of proprietary low-level APIs, most often C-based
  - Vector intrinsics
  - NVIDIA CUDA
  - ATI Brook+
  - ClearSpeed Cn
- No SW portability, hence no confidence in SW investments
  - (Brook+, Cn, Cell API are now defunct)
- De facto standards emerging (next slide)
Landscape of accelerator programming

<table>
<thead>
<tr>
<th>Interface</th>
<th>CUDA</th>
<th>OpenCL</th>
<th>DirectCompute</th>
<th>Renderscript</th>
</tr>
</thead>
<tbody>
<tr>
<td>Originator</td>
<td>NVIDIA</td>
<td>Khronos (Apple)</td>
<td>Microsoft</td>
<td>Google</td>
</tr>
<tr>
<td>Year</td>
<td>2007</td>
<td>2008</td>
<td>2009</td>
<td>2011</td>
</tr>
<tr>
<td>Area</td>
<td>HPC, desktop</td>
<td>Desktop, mobile, embedded, HPC</td>
<td>Desktop</td>
<td>Mobile</td>
</tr>
<tr>
<td>OS</td>
<td>Windows, Linux, Mac OS</td>
<td>Windows, Linux, Mac OS (10.6+)</td>
<td>Windows (Vista+)</td>
<td>Android (3.0+)</td>
</tr>
<tr>
<td>Devices</td>
<td>GPUs (NVIDIA)</td>
<td>CPUs, GPUs, custom</td>
<td>GPUs (NVIDIA, AMD)</td>
<td>CPUs, GPUs, DSPs</td>
</tr>
<tr>
<td>Work unit</td>
<td>Kernel</td>
<td>Kernel</td>
<td>Compute shader</td>
<td>Compute script</td>
</tr>
<tr>
<td>Language</td>
<td>CUDA C/C++</td>
<td>OpenCL C</td>
<td>HLSL</td>
<td>Script C</td>
</tr>
<tr>
<td>Distributed</td>
<td>Source, PTX</td>
<td>Source, SPIR</td>
<td>Source, bytecode</td>
<td>LLVM bitcode</td>
</tr>
</tbody>
</table>

- **Portability** is likely to remain an issue despite standardisation efforts
- **Performance portability** is perhaps even more of an issue!
OpenCL

- Open standard for portable programming of heterogeneous parallel systems managed by the Khronos Group
  - Apple proposed a straw-man specification (Jun-08)
  - 1.0 (Dec-08), 1.1 (Jun-10), 1.2 (Nov-11)

- Diverse industry participation
  - Processor and middleware vendors, OEMs, application developers

- New SW requirement for accelerator HW
OpenCL – programming model

- The *host* (e.g. CPU) manages a collection of *devices* (e.g. DSP + 2 GPUs) using OpenCL API calls
- Host code is compiled using a system compiler (e.g. GCC)
- Code to be executed on a device is compiled using the device’s compiler invoked at runtime
- Device code executes as a set of *work-items*, or threads
- Key idea – recast data-parallel loops as *kernel* invocations

```c
/* Vector addition in C */
void vadd (int n,
           float *c,
           const float *a,
           const float *b)
{
  for (int k = 0; k < n; ++k)
    c[k] = a[k] + b[k];
/* executed sequentially */
```

```c
/* Vector addition in OpenCL C */
__kernel void vadd (  
                       __global float *c,  
                       __global float const *a,  
                       __global float const *b)  
{  
    const int k = get_global_id(0);  
    c[k] = a[k] + b[k];
/* executed over n work-items */
```
ARCHITECTURAL TECHNIQUES
Vector processing

Single Instruction stream Multiple Data stream (SIMD)

- Parallel array of processing elements executing in lockstep
  - Solomon (c. 1962), ILLIAC IV (c. 1971), CSX (c. 2005)
- Pipelined functional units operating on data in vector registers
  - Cray-1 (c. 1976)
- Hybrid
  - Modern vector extensions
Example: parallel array execution


load \( R1, A[\text{id}] \)
load \( R2, B[\text{id}] \)
add \( R3, R1, R2 \)
store \( C[\text{id}], R3 \)

- Differently coloured cells represent different PEs
- Each PE has a unique identifier \( \text{id} \) (typically in a special register)
- Each PE has its own copy of registers \( R1, R2 \) and \( R3 \)
Example: pipelined vector execution


\begin{align*}
\text{vload} & \quad V1, A[1:4] \\
\text{vload} & \quad V2, B[1:4] \\
\text{vadd} & \quad V3, V1, V2 \\
\text{vstore} & \quad C[1:4], V3
\end{align*}

- Differently coloured cells represent different lanes of vector registers \( V1, V2 \) and \( V3 \)
- Lanes are fed one by one into pipelined functional units
Example: hybrid vector execution


- vload V1, A[1:4]
- vload V2, B[1:4]
- vadd V3, V1, V2
- vstore C[1:4], V3

\[
\begin{align*}
V1 & = \begin{bmatrix} -3 & 3 & 5 & 7 \end{bmatrix} \\
V2 & = \begin{bmatrix} 5 & -6 & 1 & 2 \end{bmatrix} \\
V3 & = \begin{bmatrix} 2 & -3 & 6 & 9 \end{bmatrix}
\end{align*}
\]

- Differently coloured cells represent different lanes of vector registers V1, V2 and V3
- Lanes are processed by parallel functional units
Vector processing (cont’d)

- Parallel, pipelined or hybrid is implementation dependent
  - Not necessarily programmer visible
- Things get more interested with conditional execution…
Example: conditional vector execution

```plaintext
if (A[1:4] > 0)
else
```

```
vload   V5, A[1:4]
vcmpeq   p, V5, 0
p.vload  V1, A[1:4]
p.vload  V2, B[1:4]
p.vadd   V3, V1, V2
p.vstore C[1:4], V3
!p.vload V4, B[1:4]
!p.vstore C[1:4], V4
```

```
A[1:4] = \begin{bmatrix} -3 & 3 & 5 & 7 \end{bmatrix}
p = V5 > 0 = \begin{bmatrix} 0 & 1 & 1 & 1 \end{bmatrix}
V1 = \begin{bmatrix} 3 & 5 & 7 \end{bmatrix}
V2 = \begin{bmatrix} -6 & 1 & 2 \end{bmatrix}
V3 = \begin{bmatrix} -3 & 6 & 9 \end{bmatrix}
V4 = \begin{bmatrix} 5 \end{bmatrix}
C[1:4] = \begin{bmatrix} 5 & -3 & 6 & 9 \end{bmatrix}
```
Fine-grained multi-threading

- Each thread has private registers inaccessible by other threads (typically a slice of a large register file)
- Scheduler switches threads on every cycle
- Having many threads in flight keeps functional units busy and allows to hide memory latency
Example: fine-grained multi-threading

Threads T1 and T2 execute statement:

\[ C[tid] = B[A[tid]] \]

I1: load R1, A[tid]
I2: load R2, B[R1]
I3: store C[tid], R2

- Each thread has a unique identifier tid (typically in a special register)
- Each thread has its own copy of registers R1 and R2
- Each thread has a unique program counter – threads can diverge

Trace (schedule):

T1.I1; R1=A[1]
T2.I1; R1=A[2]
T1.I2; R2=B[R1]
T2.I2; R2=B[R1]
T1.I2; stall
T2.I2; stall
T1.I2; stall
T2.I2; stall
T1.I2; C[2]=R2
T1.I2; stall
T1.I2; stall
T1.I2; C[1]=R2
Multi-threading + vector processing = GPU
Example: NVIDIA Tesla (CUDA)

- Instruction stream is scalar…
  - load R1, A[tid] ; coalesced load
  - load R2, B[R1] ; uncoalesced load
  - store C[tid], R2 ; coalesced store

- …but 32 threads (a warp) share the same program counter
  - Scheduler switches warps on every cycle
  - Predication achieves an illusion of independent thread execution
  - For efficiency, threads within the same warp should not diverge

- Every warp load is potentially a gather operation and every warp store is potentially a scatter operation
  - For efficiency, threads with consecutive tids should access consecutive memory locations (memory access coalescing)
Example: ARM Midgard (Mali)

- Instruction stream is vector with restricted memory accesses
  
  $\text{vload} \ V1, \ A[4*\text{tid}:4*\text{tid}+3] \ ; \ \text{vector load}$
  
  $\text{load} \ V2.x, \ B[V1.x] \ ; \ \text{gather x}$
  
  $\text{load} \ V2.y, \ B[V1.y] \ ; \ \text{gather y}$
  
  $\text{load} \ V2.z, \ B[V1.z] \ ; \ \text{gather z}$
  
  $\text{load} \ V2.w, \ B[V1.w] \ ; \ \text{gather w}$
  
  $\text{vstore} \ \ C[4*\text{tid}:4*\text{tid}+3], \ V2 \ ; \ \text{vector store}$

- ... but each thread has its own program counter
  
  - Thread divergence is not a concern
Q & A
Further reading

- “Welcome to the jungle” (Herb Sutter)
  - [http://herbsutter.com/welcome-to-the-jungle](http://herbsutter.com/welcome-to-the-jungle)

- GPU (CUDA) programming courses
  - [http://people.maths.ox.ac.uk/gilesm/cuda/lecs](http://people.maths.ox.ac.uk/gilesm/cuda/lecs) (Mike Giles)
  - [http://courses.engr.illinois.edu/ece408/lectures.html](http://courses.engr.illinois.edu/ece408/lectures.html) (Wen-mei Hwu)

- Mali developer resources
  - [http://www.arm.com/community/multimedia](http://www.arm.com/community/multimedia)
  - [http://malideveloper.arm.com](http://malideveloper.arm.com)
BACKUP SLIDES
Woes of accelerator programming

- Portability
  - I’m a Linux developer.
  - So glad I don’t have to think about DirectCompute and RenderScript.
  - OK, I’ll go with OpenCL as it’s the most portable interface.

- Usability
  - Why do I need to write so much host code just to run ‘Hello World’?
  - Phew, it’s mostly boilerplate! I’ll reuse this code for something else.
  - Now it’s time to write an interesting kernel.
  - The results are wrong. How do you mean ‘no debugging means’?
  - I need SGEMM. Do I really have to write it myself?

- Performance portability
  - My kernel runs really fast on device X but really slow on device Y?!
  - How do I optimise kernel code for different devices?
  - How do I maintain optimised code?
OPENCL MEMORY SYSTEM
OpenCL – memory system (desktop)

- Desktop systems have non-uniform memory
  - GPU is on a discrete card along with GPU (__global) memory
- Data must be physically copied between CPU (main) memory and GPU memory
  - Some algorithms take longer to perform the copying than to execute just on the CPU
OpenCL – memory system (embedded)

- Most ARM-based systems have uniform memory
  - GPU __global memory allocated in main memory (but fully cached in the GPU’s caches)
  - GPU __local memory is also allocated in main memory
- Cheap data exchange between CPU and GPU
  - Cache coherency operations are faster than physical copying