Performance and Programmability Trade-offs in the OpenCL 2.0 SVM and Memory Model

Brian T. Lewis, Intel Labs
Overview

• This talk:
  – My experience working on the OpenCL 2.0 SVM & memory models
  – Observation: tension between performance and programmability
    – Programmability = productivity, ease-of-use, simplicity, error avoidance
    – For most programmers & architects today, performance is paramount

• First, some background: why are GPUs programmed the way they are?
  – Discrete & integrated GPUs
  – GPU differences from CPUs
  – GPU performance considerations
  – GPGPU programming

• OpenCL 2.0 and a few of its features, compromises, tradeoffs
A couple of comments first

- These are my personal observations

- OpenCL 2.0, and its SVM & memory model, are the work of many people
  - I’ve been impressed by the professionalism & care paid by Khronos OpenCL members
  - Disagreements often lead to new insights
GPUs: massive data-parallelism for modest energy

- NVIDIA Tesla K40 discrete GPU: 4.3 TFLOPs, 235 Watts, $5,000
Integrated CPU+GPU processors

- More than 90% of processors shipping today include a GPU on die
- Low energy use is a key design goal

Intel 4th Generation Core Processor: “Haswell”

AMD Kaveri APU

4-core GT2 Desktop: 35 W package
2-core GT2 Ultrabook: 11.5 W package

Desktop: 45-95 W package
Mobile, embedded: 15 W package

Discrete & integrated processors

- Different points in the performance-energy design space
  - 235W vs. <1W for a GPU in a mobile SoC

- Discrete GPUs
  - Cost of PCIe transfers impacts granularity of offloading

- Integrated GPUs
  - The CPU and GPU share physical memory (DRAM)
    - Avoids cost of transferring data over a PCIe bus to a discrete GPU
  - May also share a common last-level cache
    - If so, data being offloaded is often in cache
Performance of integrated GPUs is increasing
Average speedup of 2.5x and energy savings of 2x vs. multicore CPU
GPU architecture
GPU differences from CPUs

- **CPU cores optimized for latency, GPUs for throughput**
  - CPUs: deep caches, OOO cores, sophisticated branch predictors
  - GPUs: transistors spent on many slim cores running in parallel

- **SIMT execution**
  - Work-items (logical threads) are partitioned into work-groups
  - The work-items of a work-group execute together in near lock-step
  - Allows several ALUs to share one instruction unit

Typically 256-1024 work-items per work-group

Figure by Kayvon Fatahalian, How Shader Cores Work - Beyond Programmable Shading
GPU differences from CPUs

- Shallow execution pipelines
- Low power consumption
- Highly multithreaded to hide memory latency
  - Assumes programs have a lot of parallelism
  - Switches execution to new work-group on a miss
- Separate high-speed local memory
  - Shared by work-items of an executing work-group
  - Might, e.g., accumulate partial dot-products or reduction results
- Coalesced memory accesses
  - Reduces number of memory operations
- Execution barriers
  - Synchronize work-items in work-groups

Figure by Kayvon Fatahalian, How Shader Cores Work - Beyond Programmable Shading
GPUs: but what about branches?

- Serially execute each branch path of a conditional branch
- Too much branch divergence hurts performance

Figure by Kayvon Fatahalian, From Shader Code to a Teraflop: How Shader Cores Work
For good GPU performance

- Have enough parallelism
  - Too few work-items hurts memory latency hiding

- Choose appropriate work-group size
  - Want to keep all execution units fully utilized

- Use fast local memory
  - Has low latency and high bandwidth similar to an L1 cache

- Coalesce memory accesses when possible
  - Maximize memory bandwidth

- Minimize branch divergence

Programming models tied to GPU architecture
Performance favored over programmability
– Often little performance portability
GPGPU programming
GPGPU programming: SIMT model

- CPU ("host") program often written in C or C++
  - The CPU specifies number of work-items & work-groups, launches GPU work, waits for events & GPU results
- GPU code is written as a sequential kernel in (usually) a C or C++ dialect
  - All work-items execute the same kernel
  - HW executes 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 1024x1024 image with 1,048,576 work-items

Credit: Khronos Group, OpenCL Overview
GPGPU programming: frameworks

- OpenCL
- CUDA
- C++ AMP
- Renderscript

Lower-level performance frameworks
Higher-level productivity frameworks

These differ in
- the capabilities they provide
- how much control they give programmers
- performance portability
OpenCL

- Cross-platform, cross-vendor standard for parallel & heterogeneous computing

- **Host (CPU) API**
  - Query, select, and initialize compute devices (GPU, CPU, DSP, accelerators)
  - May execute compute kernels across multiple devices

- **Kernels**
  - Basic unit of executable offloaded code
  - Built-in kernels for fixed-functions like camera pipe, video encode/decode, etc.

- **Kernel Language Specification**
  - Subset of ISO C99 with language extensions
  - Well-defined numerical accuracy: IEEE 754 rounding with specified max error
OpenCL memory & work-items

- **OpenCL 1.2:** explicit memory management
  - Application must move data from host \(\rightarrow\) global \(\rightarrow\) and back

- **Work-items/work-groups**

- **C99 kernel language restrictions**
  - No recursion since often no HW call stack
  - No function pointers

Work-group example

\[
\text{# Work-items} = \text{# pixels} \\
\text{# Work-groups} = \text{# tiles} \\
\text{Work-group size} = (\text{tile width} \times \text{tile height})
\]

http://www.slideshare.net/Khronos_Group/open-cl-overviewsiggraphasiannov13
OpenCL 2.0 changes

- **Goals:** ease of use & performance improvements

- **Shared Virtual Memory (SVM)**
  - **OpenCL 2.0:** SVM required
  - Three kinds of sharing:
    - Coarse-grain buffer sharing: pointer sharing in buffers
    - Fine-grain buffer sharing
    - Fine-grain system sharing: all memory shared with coherency
  - Fine-grain system sharing
    - Can directly use any pointer allocated on the host (malloc/free), no need for buffers
    - Both host & devices can update data using optional C11 atomics & fences

- **Dynamic Parallelism**
  - Allows a device to enqueue kernels onto itself - no round trip to host required
  - Provides a more flexible execution model
    - A very common example: kernel A enqueues kernel B, B decides to enqueue A again, ...
OpenCL 2.0 changes

• **C11 atomics**
  - Coordinate access to data accessed by multiple agents
  - Atomic loads/stores, compare & exchange, fences ...

• **OpenCL memory model**
  - With SVM and coherency, even more potential for data races
  - Based on the C11 memory model
  - Specifies which memory operations are guaranteed to happen in which order & which memory values each read operation will return
    - Supports OpenCL global/local memory, barriers, scopes, host API operations, ...
Other GPGPU frameworks

**CUDA**
- Similar to OpenCL
- Kernel language is C++ subset, no cross-device atomics
- SVM similar to coarse-grain buffer SVM
  - special allocation APIs, special pointers, non-coherent

**C++ AMP (Accelerated Massive Parallelism)**
- STL-like library for multidimensional array data
  - Runtime handles CPU<->GPU data copying
- `parallel_for_each`
  - Executes a C++ lambda at each point in an extent, tiles
- `restrict` specifies where to run the kernel: CPU or GPU

**Renderscript**
- Emphasis on mobile devices & performance portability
  - Programmer can’t control where kernels run, VM-decided
- Kernel code is C99-based
  - 1D and 2D arrays, types include size, runtime type checking
- Script groups fuse kernels for efficient invocation

**Performance**
- More control
- Often better performance

**Productivity**
- Ease of use
- Runtime checking
- More performance portability
Tradeoffs
Tradeoffs: GPGPU framework level

- Most GPGPU programs use performance frameworks
  - OpenCL, CUDA
  - Can yield best performance but more complexity, requires architectural knowledge

- Recently: growing interest in higher level, productivity frameworks
  - Renderscript aims for performance portability, does runtime type checks

- C++ AMP is between performance & productivity
  - Pragmatic, simpler framework than CUDA/OpenCL, more restricted
  - However, best performance with array tiles requires architectural knowledge

Framework design is a compromise between performance, flexibility, control and productivity, ease of use, portable performance
Tradeoffs: OpenCL 2.0 SVM

• **My opinion:** a bold decision to make this required for all 2.0 devices
  – But approved by Khronos OpenCL committee members with little discussion

• **Clear advantages...**
  – Productivity
    – SVM considerably simplifies data-structure sharing & memory management
  – Anticipated HW support for SVM
    – AMD’s Kaveri is probably just first such processor

• ...but substantial HW/SW implementation required
  – Needs page fault handling, address translation, coherency (with atomics)
  – Fine-grain system sharing (i.e. full-memory SVM) requires OS modifications
  – Maintaining coherency consumes memory bandwidth

**Trades-off implementation complexity for programmability**
Tradeoffs: OpenCL 2.0 memory scopes

- **Memory scopes: performance optimization**
  - Restricts atomic operations' effects to, e.g., just the same device
  - Scope hierarchy: work-item, work-group, device, all SVM devices

- **But what about sequential consistency?**
  - Most intuitive thread programming model
  - Can you have a single total order if all agents can't see all operations?
  - What should the default scope be for atomics?

- **Scopes impact**
  - Ease of use & understandability
  - Ease of avoiding memory errors

- **What advice do we give to (most) programmers?**
  - When is sequential consistency guaranteed?

**Classic performance-ease of use tradeoff**
Tradeoffs: Consume ordering in OpenCL 2.0

- **C11 & C++11 have a consume memory order for atomics & fences**
  - Can improve performance on certain architectures: e.g., ARM & Power
  - Provide guarantees about sequencing operations based on tracking value dependencies
  - On most architectures, can be implemented as acquire with no loss of performance

- **But this adds complexity — visible in C/C++11 memory models**
  - Extra dependency-ordered-before & inter-thread-happens-before relations
  - Is keeping closer to the C11 model worth the added complexity?

- **OpenCL committee approved dropping consume**
  - Useful on few GPUs

---

Trades-off backwards compatibility & (possible) performance for programmability
Conclusions

- There is a tension between performance & programmability
  - Historically, programming models tied to GPU architecture
  - Performance more important than programmability
- But signs of change
  - Perhaps driven by desire to increase use of GPUs & to improve performance/Watt
  - Support for SVM, atomics, coherency, Renderscript's automatic work placement
  - Growing interest in higher level, productivity frameworks
Traditional GPU software stack

Driver does:
- command validation
- memory reference validation
- argument patching
- scheduling commands

Result: fixed minimum kernel launch overhead

Offloading cost impacts offload granularity

3/2/2014

Trade-offs in OpenCL 2.0 SVM and Memory Model
OpenCL basics: executing programs

1. Query for OpenCL devices
2. Create context for selected devices
3. Select kernels
4. Create memory objects
5. Copy memory objects to devices
6. Enqueue kernels for execution
7. Copy kernel results back to host

http://www.slideshare.net/Khronos_Group/open-cl-overviewsiggraphasianov13

3/2/2014 Trade-offs in OpenCL 2.0 SVM and Memory Model
GPGPU programming frameworks

![Diagram showing performance vs productivity for GPGPU programming frameworks including CUDA, OpenCL, C++ AMP, OpenACC, and Renderscript.]
NVIDIA CUDA

- Popular GPGPU framework, Similar to OpenCL
- Like OpenCL:
  - SVM with CUDA Unified Virtual Memory
    - Somewhat like OpenCL's coarse-grain buffer sharing, no coherency, avoids manual data copying
    - Uses special virtual memory pointers, specialized allocation APIs
  - Device self-enqueuing of kernel invocations
  - Device-to-CPU fences: __threadfence_system()
- Differences from OpenCL:
  - Host & kernel code in same source file, NVCC compiler
  - Kernel code is C++ subset
    - Includes virtual methods, function pointers (to device functions)
    - No exceptions, RTTI, C++ Standard Library
  - Device malloc/free
  - Atomics are only atomic on same device
• **Microsoft’s C++ AMP (Accelerated Massive Parallelism)**
  - Part of Visual C++, integrated with Visual Studio, built on Direct3D
  - “Performance for the mainstream”

• **STL-like library for multidimensional array data**
  - Special convenience support for 1, 2, and 3 dimensional arrays on CPU or GPU
  - C++ AMP runtime handles CPU<->GPU data copying
  - Tiles enable efficient processing of sub-arrays
    - Essentially matches sub-arrays with work-groups to process them

• **parallel_for_each**
  - Executes a kernel (C++ lambda) at each point in the extent
  - restrict() clause specifies where to run the kernel: cpu (default) or direct3d (GPU)
    - Typical requirements for C++ code of amp kernels: no virtual methods, function pointers, …
    - In future, might have specifiers for pure (side-effect free) & write-only code
void AddArrays(int n, int * pA, int * pB, int * pC) {
    array_view<int,1> a(n, pA);
    array_view<int,1> b(n, pB);
    array_view<int,1> sum(n, pC);

    parallel_for_each( sum.grid, [=](index<1> idx) mutable restrict(direct3d) {
        sum[idx] = a[idx] + b[idx];
    });
}
C++ AMP at a Glance

- restrict(direct3d, cpu)
- parallel_for_each
- class array<T,N>
- class array_view<T,N>
- class index<N>
- class extent<N>
- class grid<N>
- class accelerator
- class accelerator_view
- class tiled_grid<Z,Y,X>
- class tiled_index<Z,Y,X>
- class tile_barrier
- tile_static storage class
Renderscript

• Higher-level than CUDA or OpenCL: simpler & less performance control
  - Emphasis on mobile devices & cross-SoC performance portability

• Programming model
  - C99-based kernel language, JIT-compiled, single input-single output
  - Automatic Java class reflection
  - Intrinsics: built-in, highly-tuned operations, e.g. ScriptIntrinsicConvolve3x3
  - Script groups combine kernels to amortize launch cost & enable kernel fusion

• Data type:
  - 1D/2D collections of elements, C types like int and short2, types include size
  - Runtime type checking

• Parallelism
  - Implicit: one thread per data element, atomics for thread-safe access
  - Thread scheduling not exposed, VM-decided
OpenACC

Automatically maps compute-intensive loops to accelerators
- Supports either vector or parallel accelerators, e.g. GPUs and Xeon Phi
- OpenACC compilers manage offloading & data movement based on directives/pragmas
  - Compilers from CAPS enterprise, Cray, and The Portland Group (PGI)/NVIDIA
- Works with existing HPC programming models like OpenMP, MPI, CUDA & OpenCL

Some key C++ directives for C++ (similar ones for Fortran)
- `#pragma acc kernels [clause [[,] clause]...} { structured block }`
  - Defines a program region to be compiled into one or more kernels
- `#pragma acc loop [clause [[,] clause]...} statement`
  - The clauses specify how to accelerate the following loop: e.g., `gang(64)`
- `copy(list), copyin(list), and copyout(list)`
  - Copy specified data to & from the accelerator
void convolution_SM_N(typeToUse A[M][N], typeToUse B[M][N])
{
    int i, j, k;
    int m=M, n=N;

    // Compile following region into a sequence of kernels
    #pragma acc kernels pcopyin(A[0:m]) pcopy(B[0:m])
    {
        double c11, c12, c13, c21, c22, c23, c31, c32, c33;
        c11 = +2.0f;  c21 = +5.0f;  c31 = -8.0f;
        c12 = -3.0f;  c22 = +6.0f;  c32 = -9.0f;
        c13 = +4.0f;  c23 = +7.0f;  c33 = +10.0f;

        // Execute the loop iterations in parallel across a number of gangs
        #pragma acc loop gang(64)
        for (int i = 1; i < M - 1; ++i) {
            // Execute the loop in parallel using the specified workers within the gangs
            #pragma acc loop worker(128)
            for (int j = 1; j < N - 1; ++j) {
                + c21 * A[i-1][j+0]  +  c22 * A[i+0][j+0]  +  c23 * A[i+1][j+0]
                + c31 * A[i-1][j+1]  +  c32 * A[i+0][j+1]  +  c33 * A[i+1][j+1];
            }
        }
    } // kernels region
}
HSA

- **Heterogeneous System Architecture from the HSA Foundation**
  - Key members: AMD, QUALCOMM, ARM, SAMSUNG, TI

- **System architecture easing efficient use of accelerators, SoCs**
  - Intended to support high-level parallel programming frameworks
    - E.g., OpenCL, C++ AMP, C++, C#, OpenMP, Java
  - Accelerator requirements
    - Full-system SVM, memory coherency, preemption, user-mode dispatch
  - Portable low-level compiler IR: HSAIL
    - Supports all of OpenCL & C++ AMP

Many HSA member companies are also active with Khronos in the OpenCL™ working group.

Trade-offs in OpenCL 2.0 SVM and Memory Model