Concord: Homogeneous Programming for Heterogeneous Architectures

Rajkishore Barik, Intel Labs
Brian T. Lewis, Intel Labs

Presenters

- **Rajkishore Barik, Intel Labs**
  - Research Scientist, Programming Systems Lab
  - Interests: Compilers and runtimes for heterogeneous architectures
  - Ph.D: Rice University
- **Brian T. Lewis, Intel Labs**
  - Senior Research Scientist, Programming Systems Lab
  - Interests: Programming language implementation, virtual machines for managed programming languages, runtime systems, and heterogeneous computing
  - Ph.D: University of Washington

Tutorial Motivation & Goals

- **Motivation**
  - GPUs should be programmed the same way as multi-core CPUs
    - No new language invention
  - How can we enable existing multi-core software stack to take advantage of integrated GPUs with minimal effort?

- **Goals**
  - Introduction to GPGPU programming framework
  - Concord C++ programming model

What this tutorial will cover

- GPGPU architectures
  - Discrete accelerators
  - Integrated accelerators
- Existing GPGPU programming models
  - CUDA, OpenCL, C++ AMP, Renderscript
- Concord C++ programming model
  - Language constructs and restrictions
  - Compiler and runtime framework
    - Virtual Functions and Reductions on GPUs
    - Compiler optimizations
  - Experimental evaluation
- Demonstration
- Q&A

Tutorial Slides

- **Available at https://github.com/IntelLabs/iHRC/docs**
- **Concord research framework is available at https://github.com/IntelLabs/iHRC/**

Overview of GPUs and GPGPU Programming

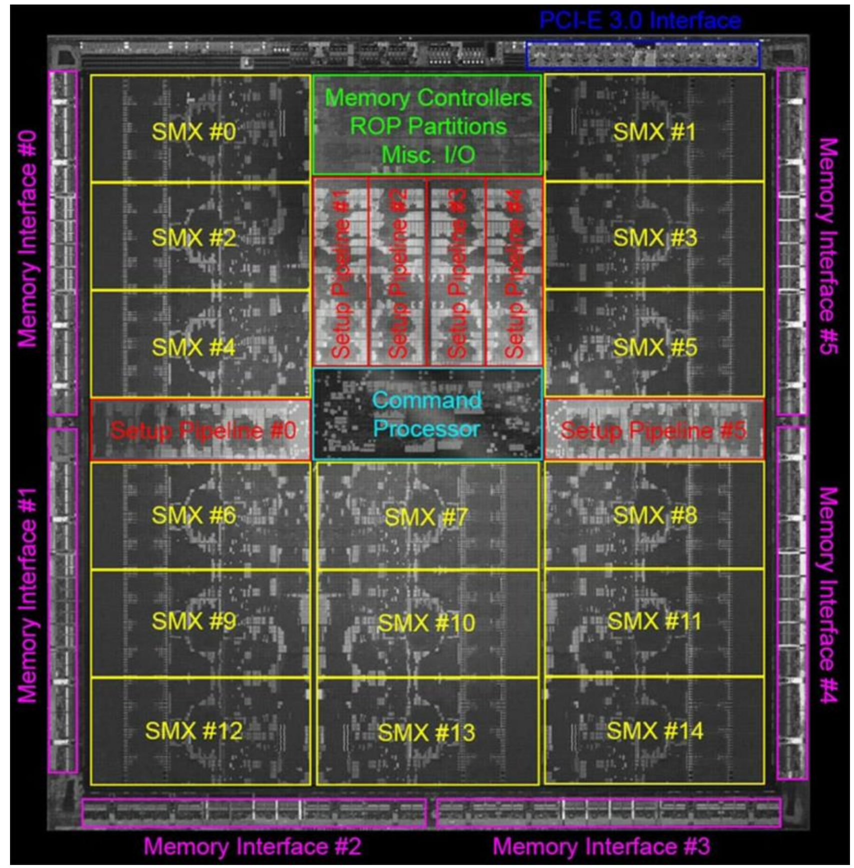
Brian T. Lewis, Intel Labs

GPUs: massive data-parallelism for little energy

- **NVIDIA Tesla K40 discrete GPU: 4.3 TFLOPs, 235 Watts, $5,000**

Features	Tesla K40
Number and Type of GPU	1 Kepler GK110B
Peak double precision floating point performance	1.43 Tflops
Peak single precision floating point performance	4.29 Tflops
Memory bandwidth (ECC off)	288 GB/sec
Memory size (GDDR5)	12 GB
CUDA cores	2880

ALUs or "lanes"

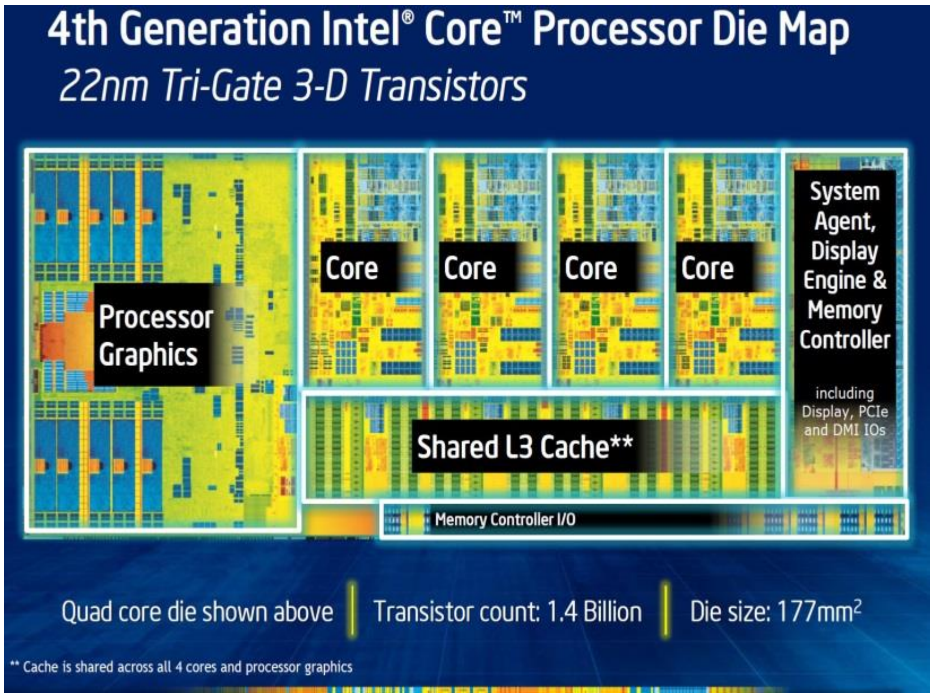


http://forum.beyond3d.com/showpost.php?p=1643034&postcount=107

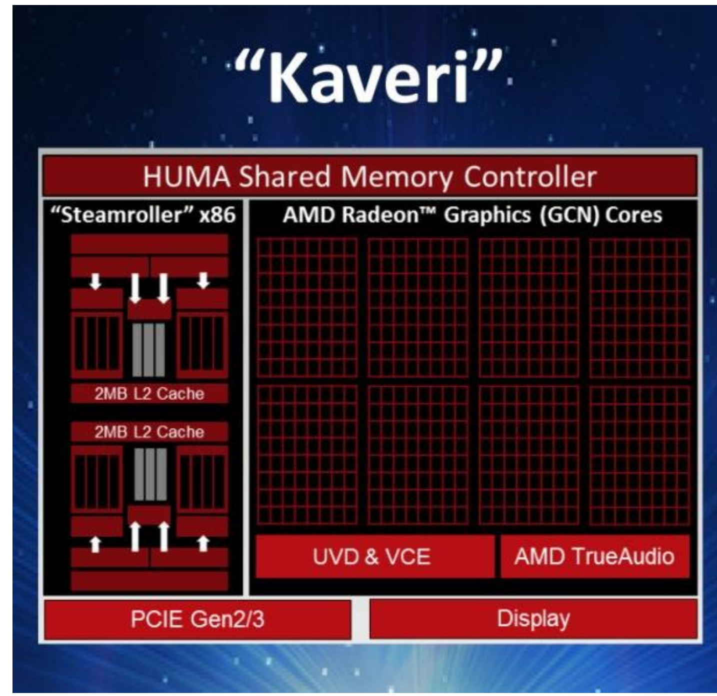
Integrated CPU+GPU processors

- **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



http://www.geeks3d.com/20140114/amd-kaveri-a10-7850k-a10-7700k-and-a8-7600-apus-announced/

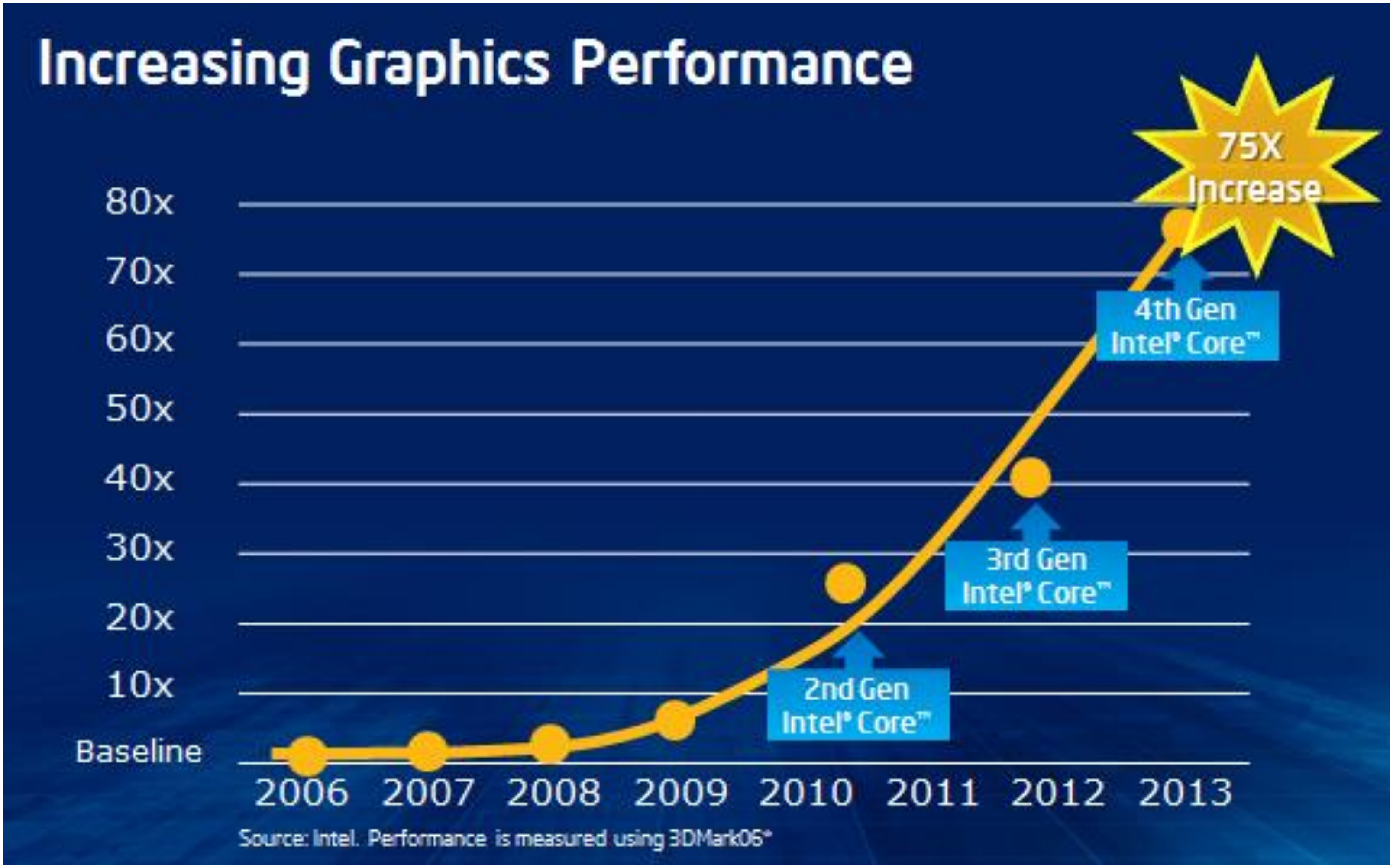
4-core GT2 Desktop: 35 W package
2-core GT2 Ultrabook: 11.5 W package

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

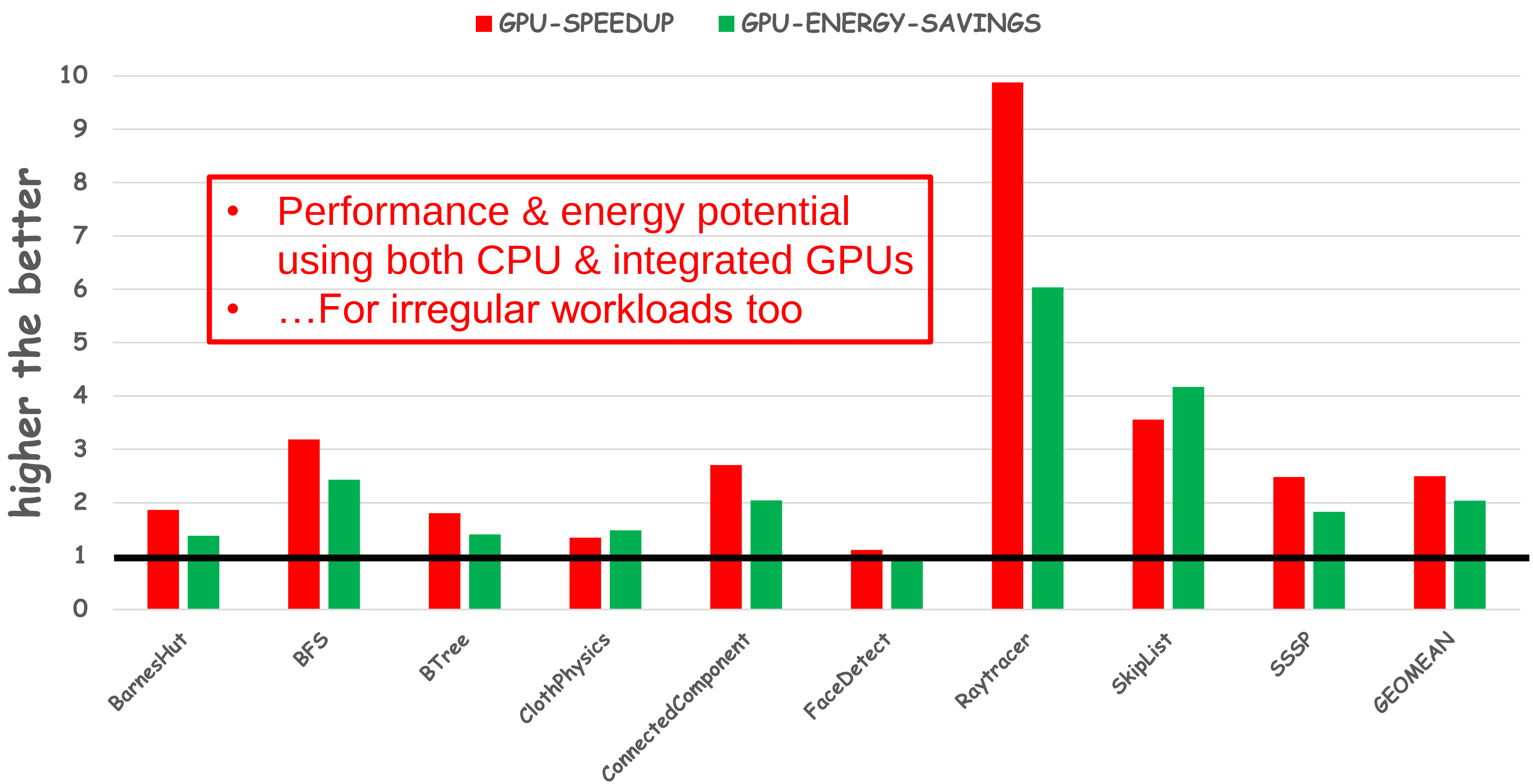
Discrete & integrated processors

- **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
    - E.g., Intel Core processors share the last-level cache
    - Data being offloaded is often in cache

Increasing performance of integrated GPUs



Ultrabook: Speedup & Energy savings compared to multicore CPU

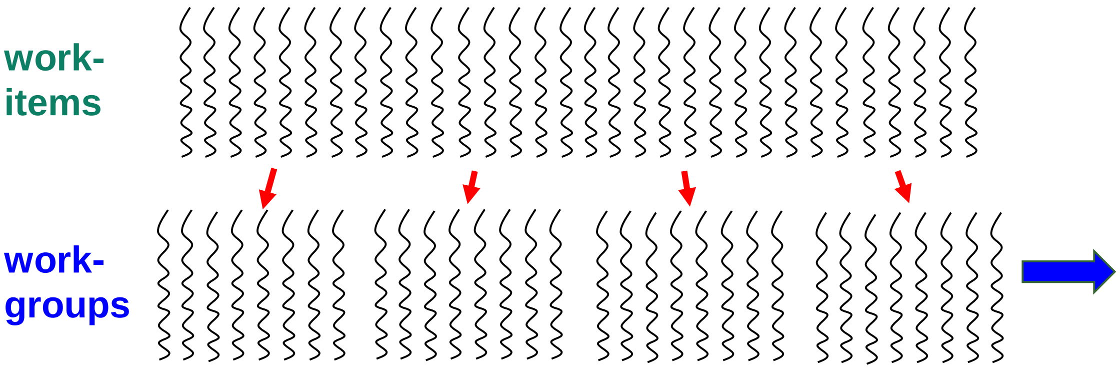


Average speedup of 2.5x and energy savings of 2x vs. multicore CPU

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



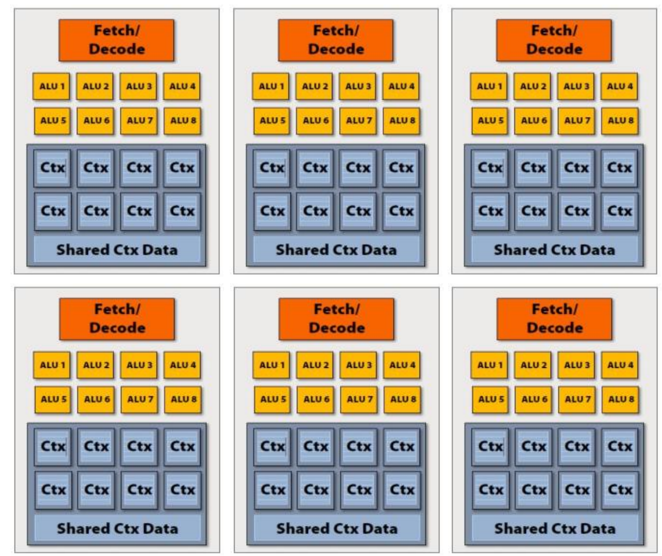


Figure by Kayvon Fatahalian, How Shader Cores Work – Beyond Programmable Shading

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

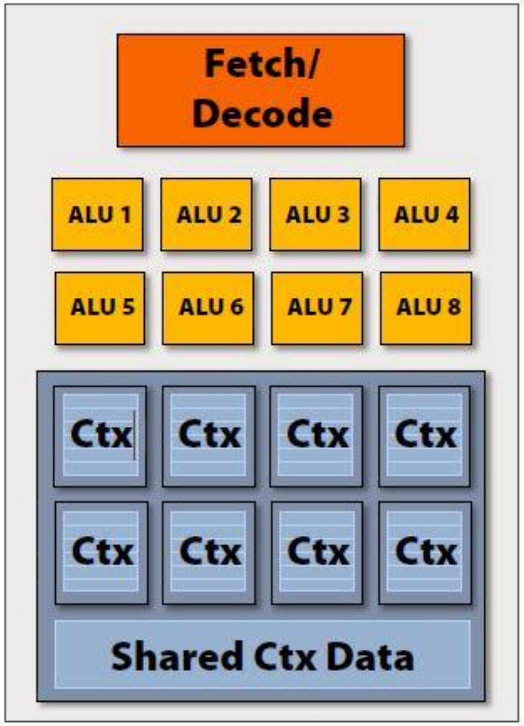


Figure by Kayvon Fatahalian, How Shader Cores Work – Beyond Programmable Shading

GPUs: but what about branches?

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

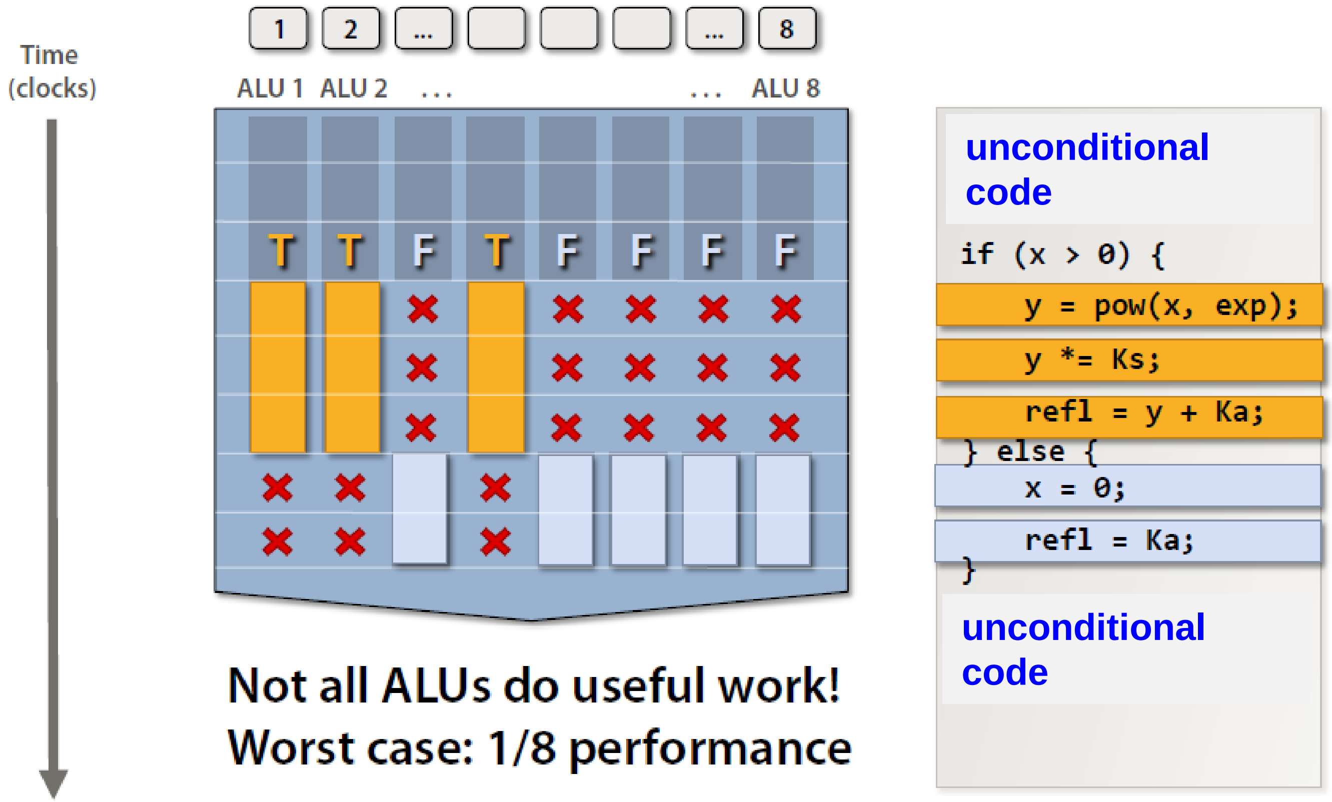


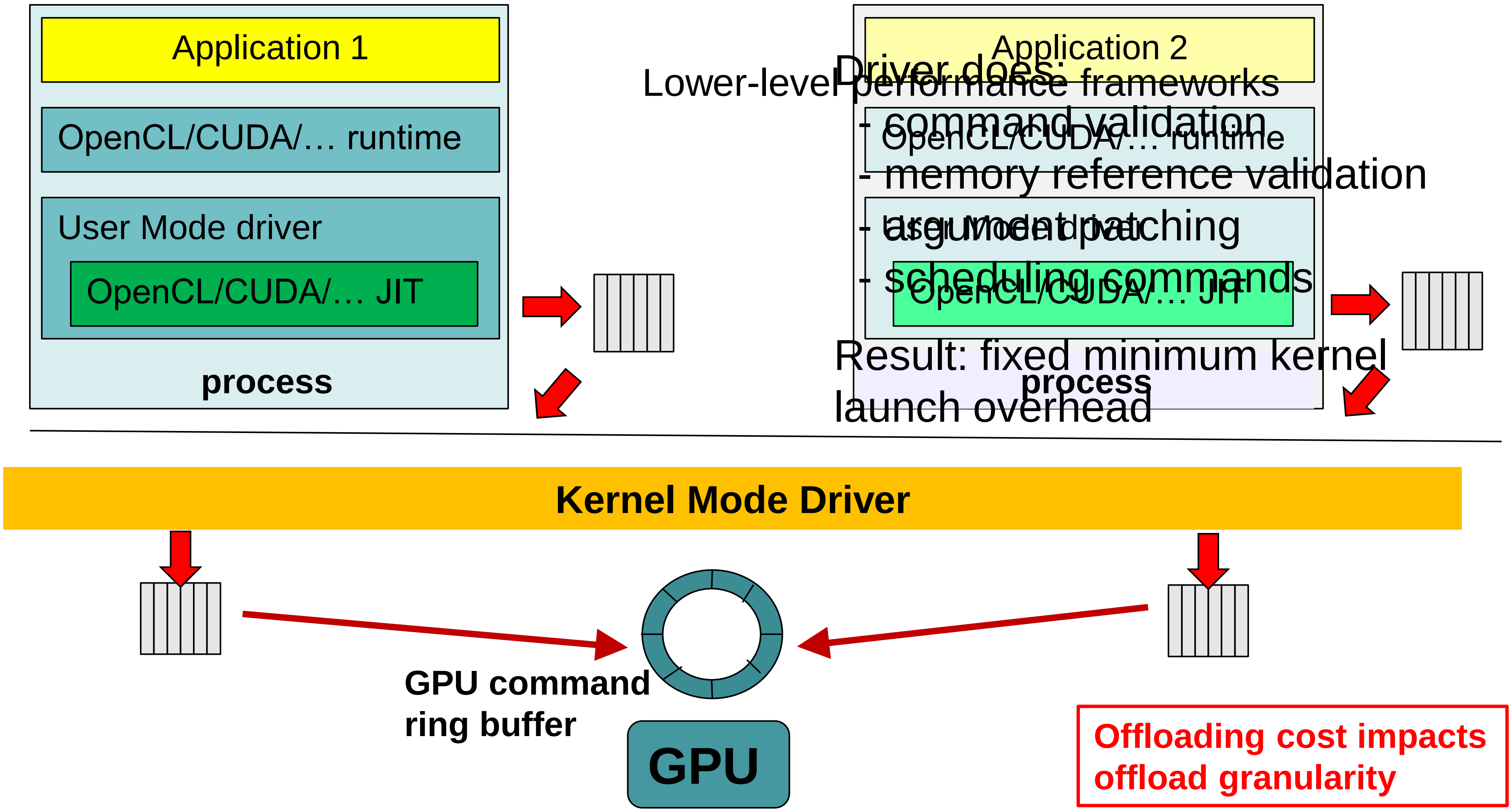
Figure by Kayvon Fatahalian, From Shader Code to a Teraflop: How Shader Cores Work

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**

Traditional GPU software stack



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

E.g., process 1024x1024 image with 1,048,576 work-items

Traditional loops

```
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

```
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
```

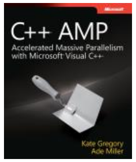
Credit: Khronos Group, OpenCL Overview

GPGPU programming: frameworks

- **OpenCL**
- **CUDA**

Lower-level performance frameworks

- **C++ AMP**



- **Renderscript**

Higher-level productivity frameworks

- **HSA**

System architecture for efficient accelerator use

OpenCL

- **Cross-platform, cross-vendor standard for parallel & heterogeneous computing**
- **C Platform Layer API on host (CPU)**
  - Query, select. and initialize compute devices (GPU, CPU, DSP, accelerators)
  - Execute compute kernels across multiple devices
- **Kernel**
  - 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
  - Rich set of built-in functions: dot, sin, cos, pow, log …

OpenCL memory & work-items

- **Memory management is often explicit**
  - Application must move data from host → global → and back
- **Command queue**
  - Used to enqueue kernels & data transfers
  - Performed in-order or out-of-order
- **Work-items/work-groups**
- **C99 kernel language restrictions**
  - No recursion since often no HW call stack
  - No function pointers

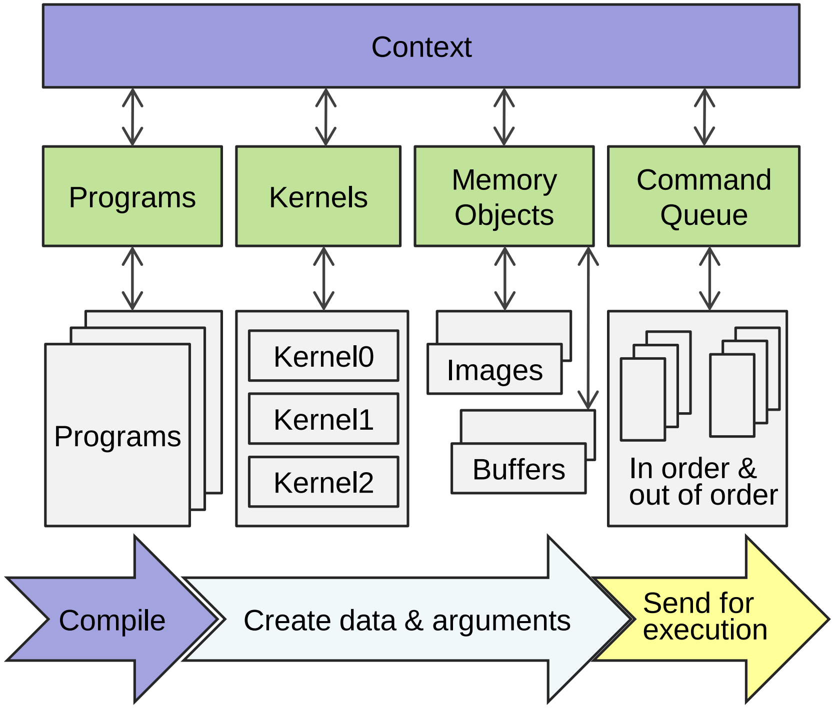
Work-group example

Work-items = # pixels
Work-groups = # tiles
Work-group size = (tile width * tile height)

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

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

OpenCL 2.0 changes

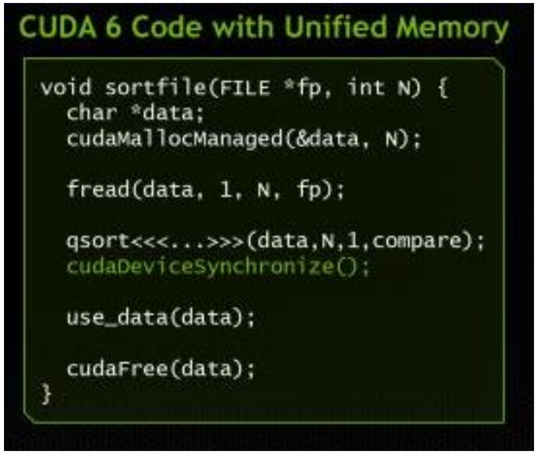
- **Shared Virtual Memory (SVM)**
  - OpenCL 2.0: SVM required
  - Three kinds of sharing:
    - Coarse-grain buffer sharing: pointer sharing in buffers
    - Fine-grain buffer sharing
    - Fine-grain system sharing: all memory shared with coherency
  - Fine-grain system sharing
    - Can directly use 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

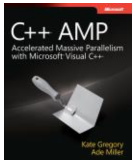
- **C11 atomics**
  - Coordinate access to data accessed by multiple work-items & host threads
    - Atomic loads/stores, compare & exchange, read-modify-write operations, fences …
  - Support for OpenCL features like global/local memory, memory scopes, …
- **OpenCL memory model**
  - With SVM and coherency, even more potential for data races
  - Based on C11 memory model
  - Specifies which memory operations are guaranteed to happen in which order & which memory values each read operation will return
    - Defines data race, when atomics synchronize, when sequential consistency is guaranteed, …
    - Supports OpenCL barriers, global/local memory, scopes, host API operations, …

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



C++ AMP



- **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

Basic Elements of C++ AMP coding

parallel_for_each: execute lambda on the accelerator once per thread

```
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];
        }
    );
}
```

restrict(direct3d): tells the compiler to check that this code can execute on DirectX hardware

array_view: wraps the data to operate on the accelerator

grid: the number and shape of threads to execute the lambda

array_view variables captured and copied to device (on demand)

index: the thread ID that is running the lambda, used to index into captured arrays

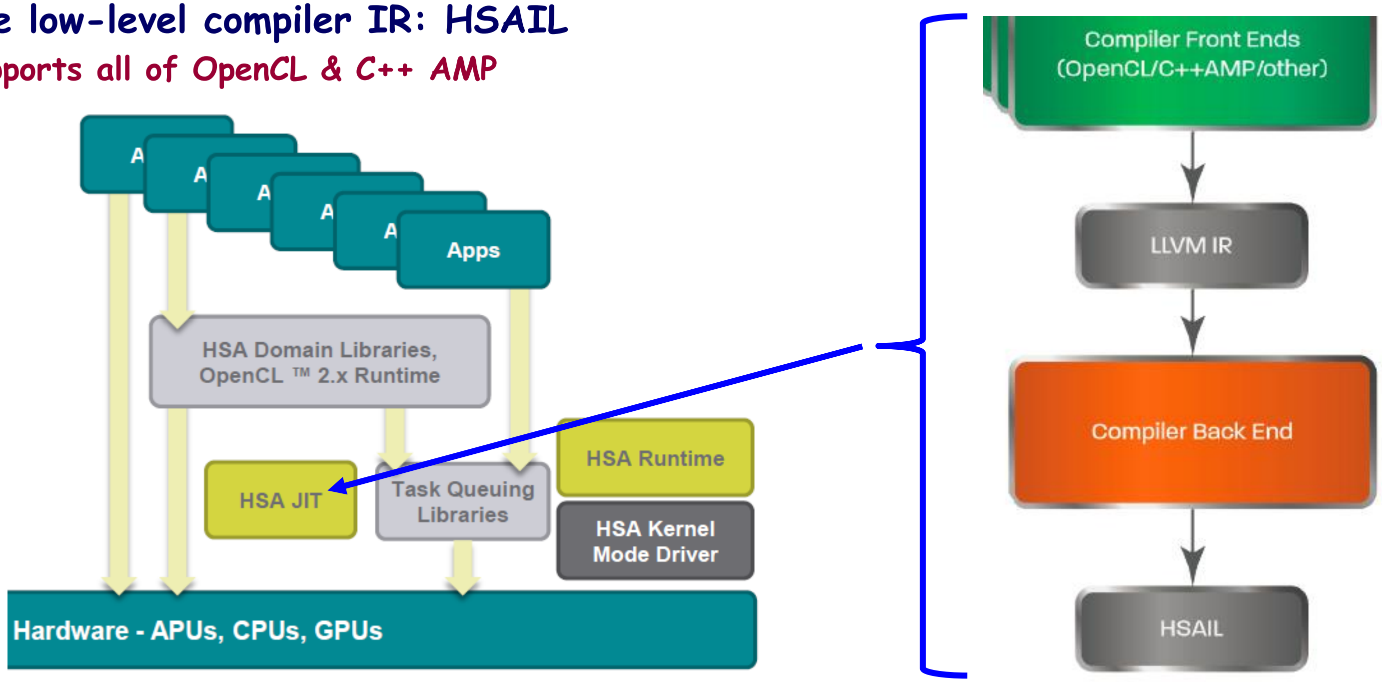
Renderscript

- **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

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



Backup

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

OpenACC

OpenACC® DIRECTIVES FOR ACCELERATORS

- **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

OpenACC

OpenACC DIRECTIVES FOR ACCELERATORS

```
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) {
        B[i][j] = c11 * A[i-1][j-1]  +  c12 * A[i+0][j-1]  +  c13 * A[i+1][j-1]
                + 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
}
```