OpenCL Introduction

PRACE/LinkSCEEM Winter School January, 26th 2011

Agenda

- Introduction
- OpenCL Architecture
- OpenCL Application

Introduction

Before OpenCL

- GPGPU
  - Vertex / pixel shaders
  - Heavily constrained and not adapted
- Brook
  - Then Brook+
  - Then CAL/IL
- CUDA
  - Widely broadcasted
- No one of these technologies is hardware agnostic
  - Portability is not possible

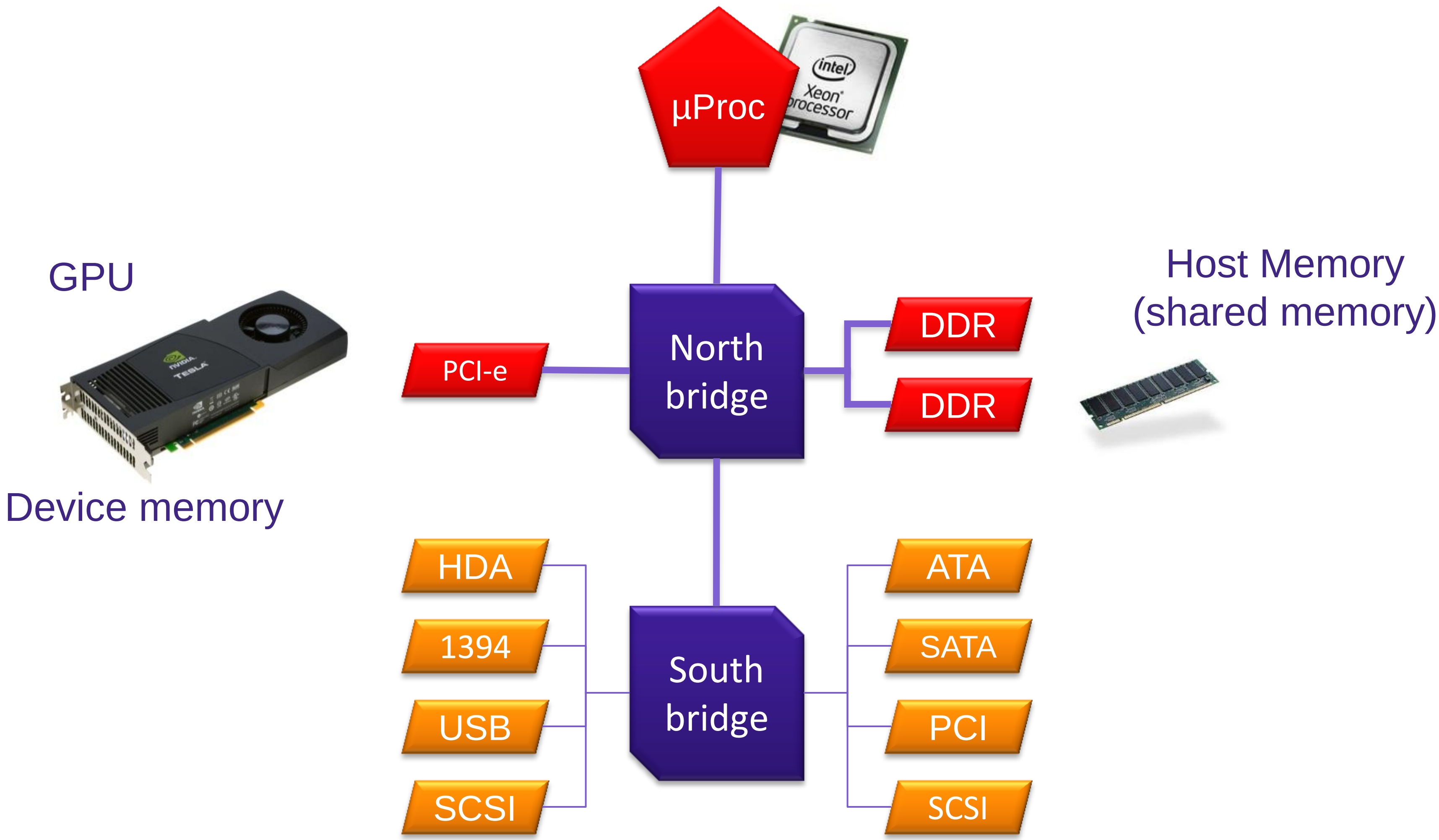
What is Hybrid Computing with OpenCL?

- OpenCL is
  - Open, royalty-free, standard
  - For cross-platform, parallel programming of modern processors
  - An Apple initiative
  - Approved by Intel, Nvidia, AMD, etc.
  - Specified by the Khronos group (same as OpenGL)

- It intends to unify the access to heterogeneous hardware accelerators
  - CPUs (Intel i7, …)
  - GPUs (Nvidia GTX & Tesla, AMD/ATI 58xx, …)

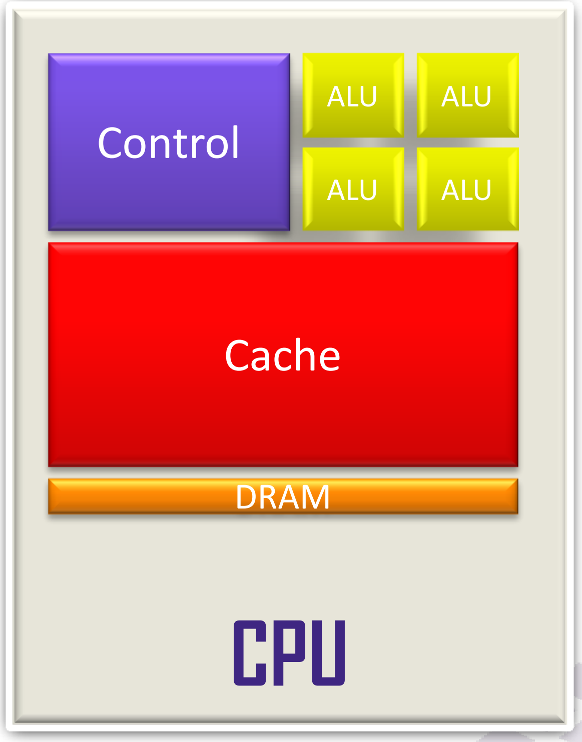
- What's the difference with CUDA or CAL/IL?
  - Portability over Nvidia, ATI, S3… platforms + CPUs

Heterogeneous Platforms Architecture

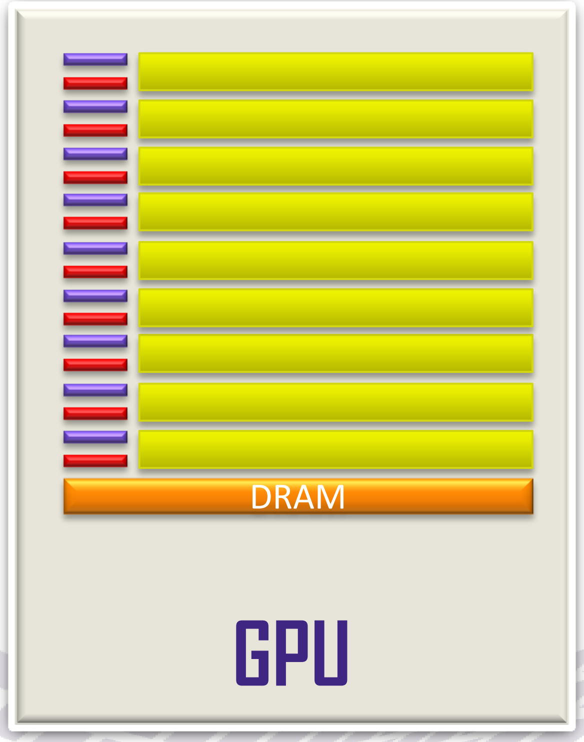


CPU vs GPU Architecture

- Computational kernels would be different



General purpose architecture



Massively data parallel

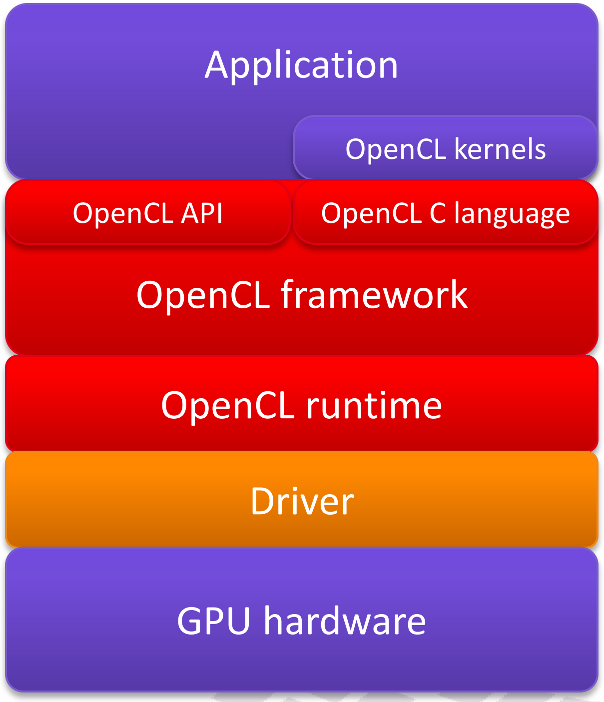
OpenCL Devices

- Intel & AMD
  - X86 w/ >= SSE 3.x
- S3
  - NV1000
  - 5400E
  - …
- IBM Cell
- NVIDIA
  - All CUDA cards
  - But not all the drivers
- ATI
  - Radeon & Radeon HD
  - FirePro, FireStream
  - Mobility…



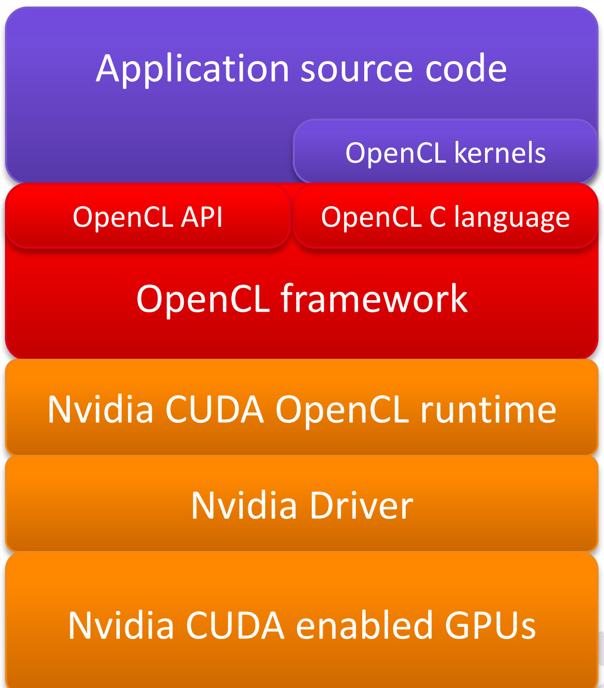
Inputs/Outputs with OpenCL programming

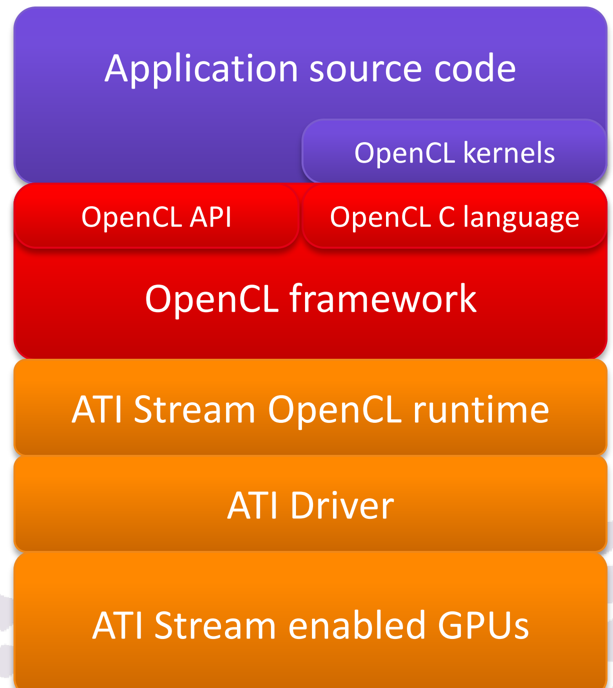
- OpenCL architecture



Inputs/Outputs with OpenCL programming

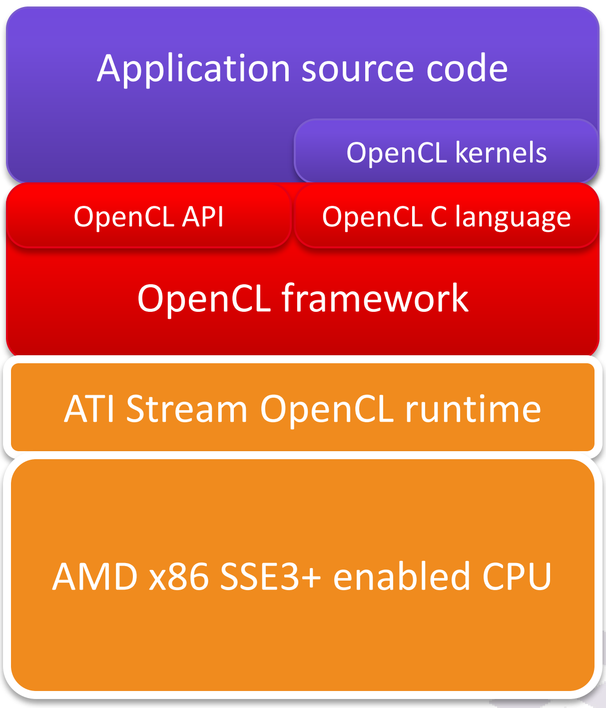
- Nvidia GPUs
- Ati GPUs

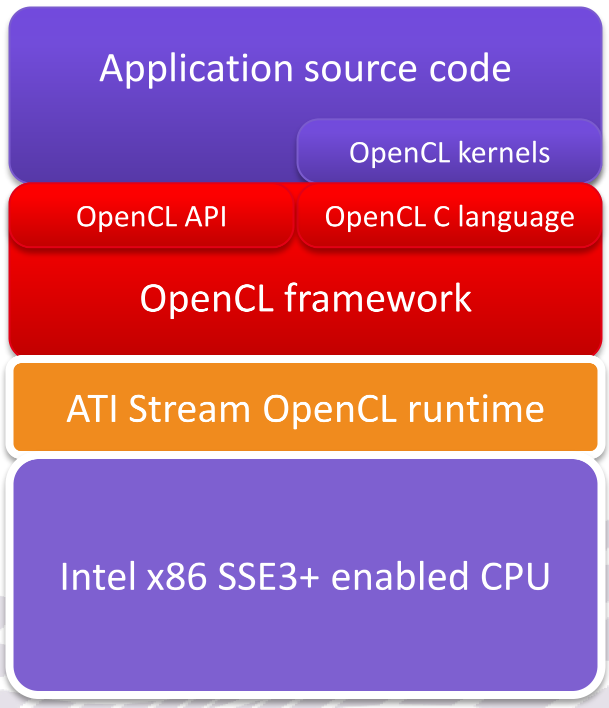




Inputs/Outputs with OpenCL programming

- AMD processors
- Intel processors

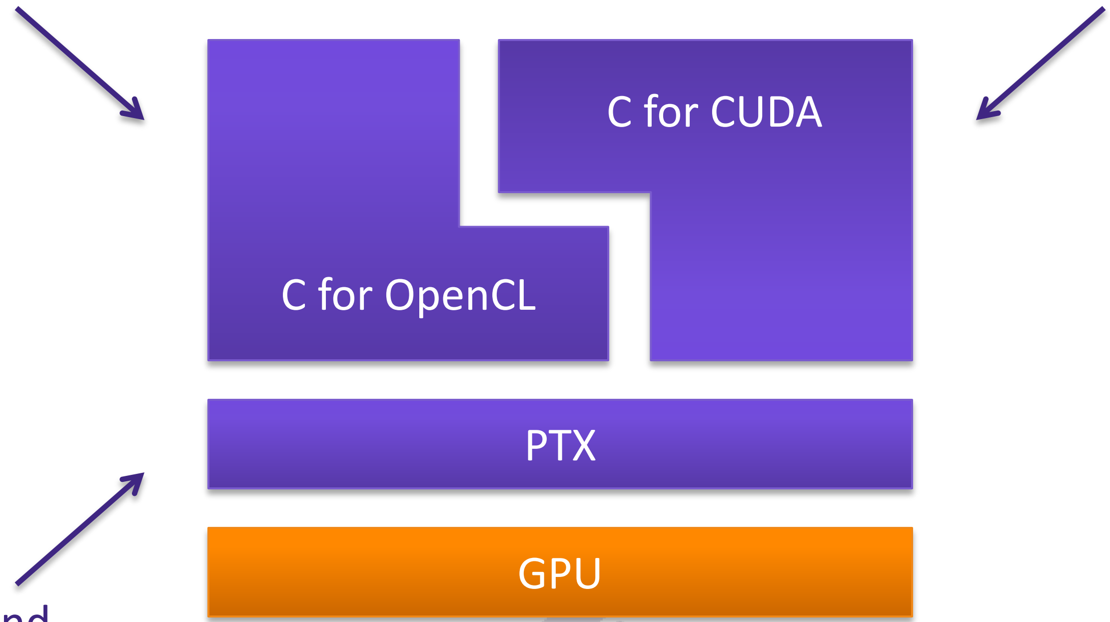




OpenCL and C for CUDA

Entry point for developers who prefer low level API

Entry point for developers who prefer high level C



Shared backend compiler and optimization technology

OpenCL APIs

- Compilation and execution
  - Include header and link with OpenCL library
  - On ATI, export DISPLAY=:0.0

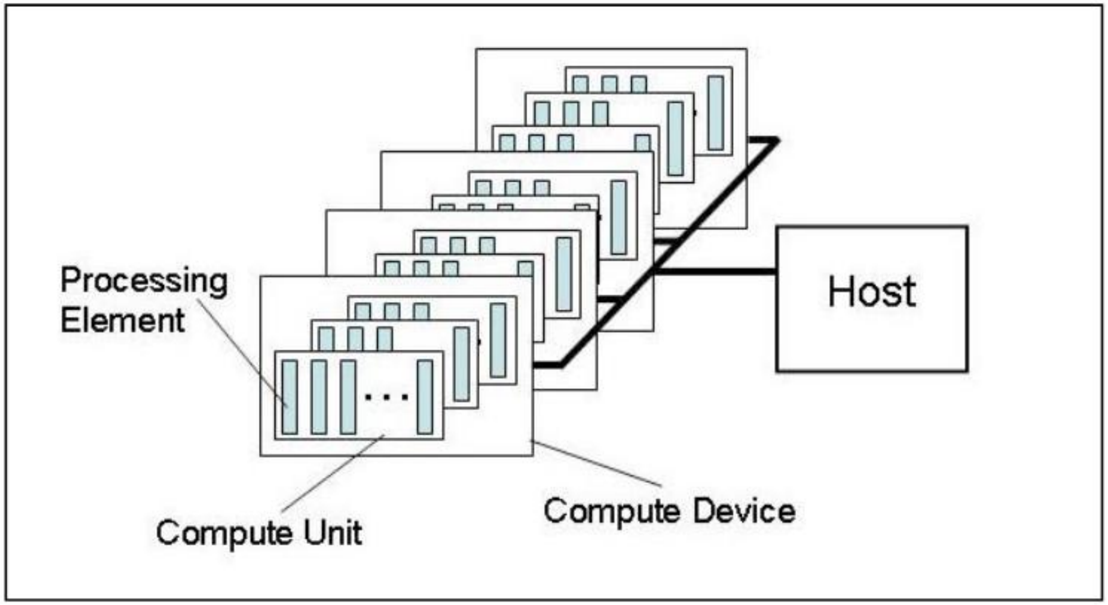
- C language API
  - Bindings C++ (official)
  - Bindings Java
  - Bindings Python
  - …

- Extensions exist to
  - OpenGL
  - Direct3D

OpenCL Architecture

Platform Model

- Model consists of one or more interconnected devices



- Computations occur within the Processing Elements of each device

Platform Version

- 3 different kind of versions for an OpenCL device
- The platform version
  - Version of the OpenCL runtime linked with the application
- The device version
  - Version of the hardware (driver)
- The language version
  - Higher revision of the OpenCL standard that this device supports

Execution Model

- Kernels are submitted by the host application to devices throw command queues
- Kernel instances, called Work-Item (WI), are identified by their point in the NDRange index space
  - This enables to parallelize the execution of the kernels

- But still 2 programming models are supported
  - Data-parallel
  - Task parallel

- So even if we have a single programming model, we should have two different programming approaches according to the paradigm we are considering

Programming Model

- Data-parallel
  - A frequent way is a one-to-one mapping between the elements of a memory object and the WI space a kernel can implement in parallel
- Task parallel
  - Single instance of kernels are executed independly of any NDRange
  - Equivalent to work-group contains only one work-item
- The programmer then defines the number of work-items to execute and their allocation into work-groups
- 2 ways of synchronization
  - On the hardware: WI of a same WG can synchronize their execution
  - On the host: commands queued up to the Command Queue for a same context can be executed asynchronously, in order or not

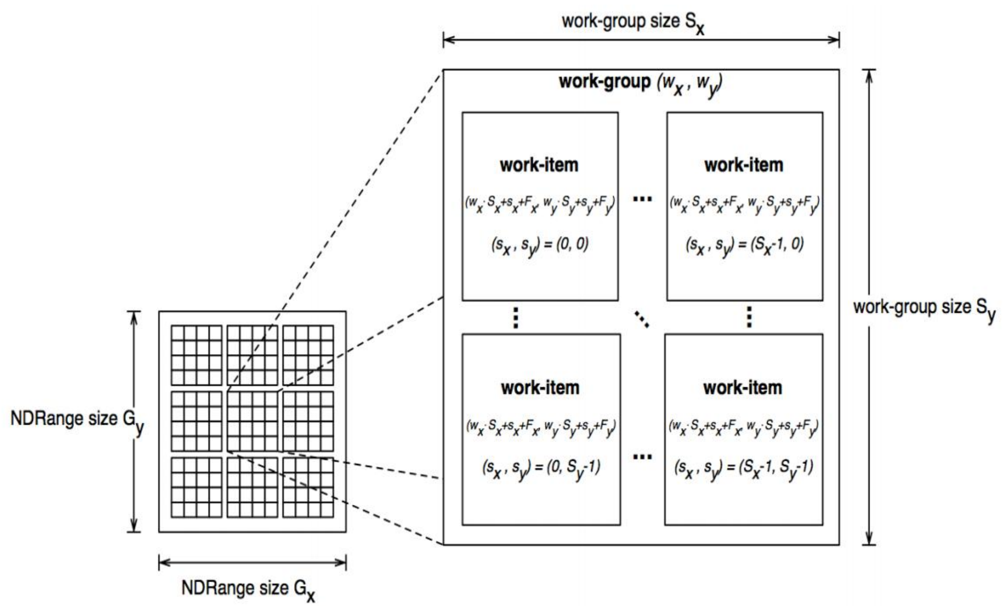
Parallelism Grains

- CPU cores can handle only a few tasks
  - But more complex
    - Hard control flows
    - Memory cache
  - Different tasks
  - Or small computation grid (NDRange)

- GPU threads are extremely lightweight
  - Very little creation overhead
  - Simple and regular computations
  - GPU needs 1000s of threads (w.i.) for full efficiency

NDRange

- NDRange is a N-Dimensional index space
  - N is 1, 2 or 3
  - NDRange is defined by an integer array of length N specifying the extent of the index space on each dimension



Work-Groups & Work-Items

- Work-Items are organized into Work-Groups (WG)

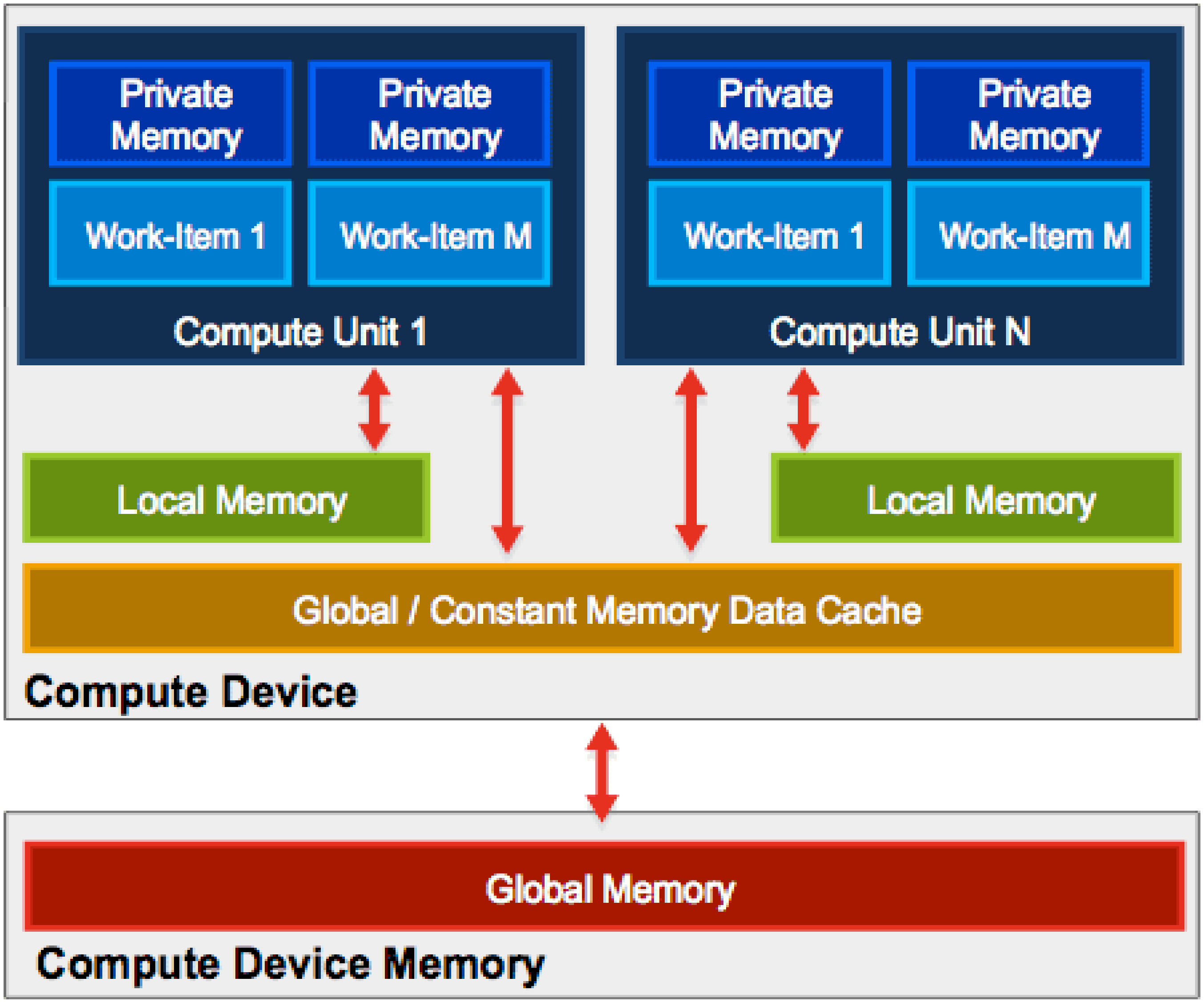
- Each Work-group has a unique global ID in the NDRange

- Each Work-item has
    - A unique global ID in the NDRange
    - A unique local ID in his work-group

Memory Model

- Four distinct memory regions
  - Global Memory
  - Local Memory
  - Constant Memory
  - Private Memory
- Global and Constant memories are common to all WI
  - May be cached depending on the hardware capabilities
- Local memory is shared by all WI of a WG
- Private memory is private to each WI

Memory Architecture



Memory Allocation

- Allocations depends on user's point of view

	Global	Constant	Local	Private
Host	Dynamic Allocation R/W	Dynamic Allocation R/W	Dynamic Allocation No Access	No Allocation No Access
Kernel	No Allocation R/W	Static Allocation Read-only	Static Allocation R/W	Static Allocation R/W

Memory Space Qualifier

- Qualifiers
  - __globlal for global memory
  - __local for local memory
  - __constant for constant memory
  - __private for private memory

- The "__" prefix is not required before the qualifiers

- Inside a kernel
  - By default variables are in private memory
  - By default, variables declared as pointer points to private memory

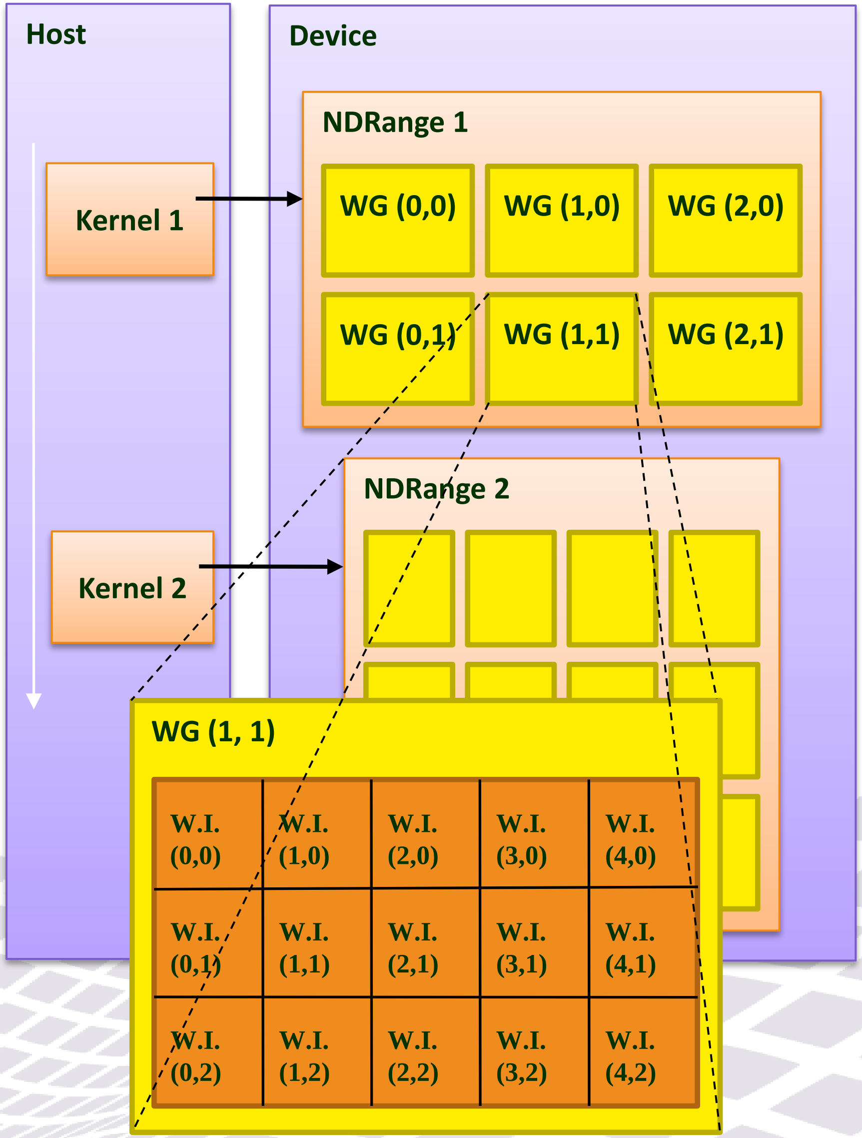
- Pointers passed as arguments to a kernel function must be of type __global, __constant, or __local

Thread Batching: Work Groups and NDRange

- A kernel is executed as a grid of work groups (NDRange)
  - All W.I.can share a data memory space insinde their work group

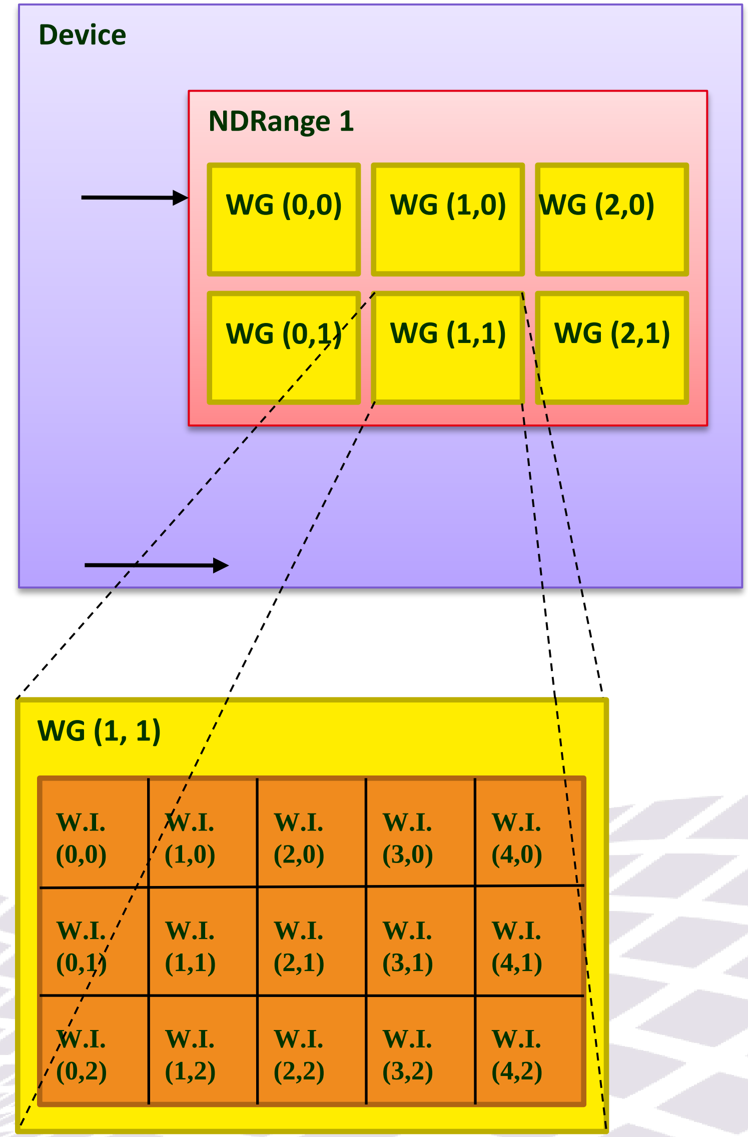
- A work group is a batch of threads which can cooperate with each other by:
  - Synchronizing their execution
    - For hazard-free local memory accesses
  - Efficiently sharing data through a low latency local memory

- Two threads from two different WG cannot cooperate



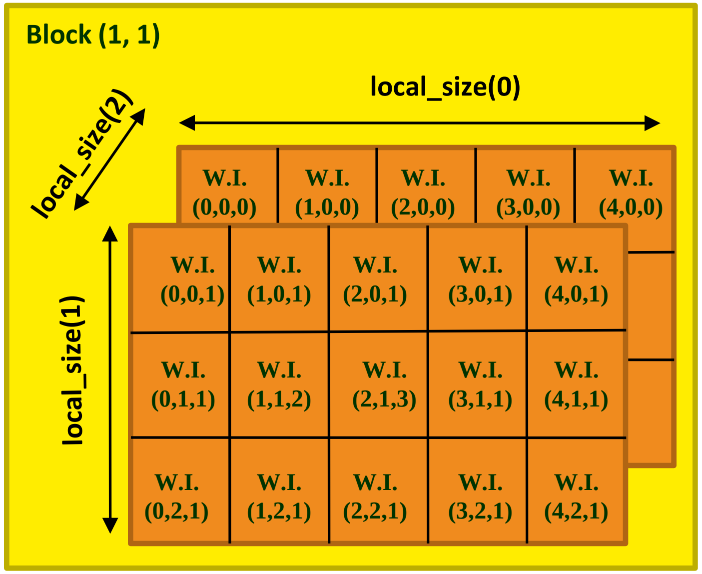
Work groups and Work Item IDs

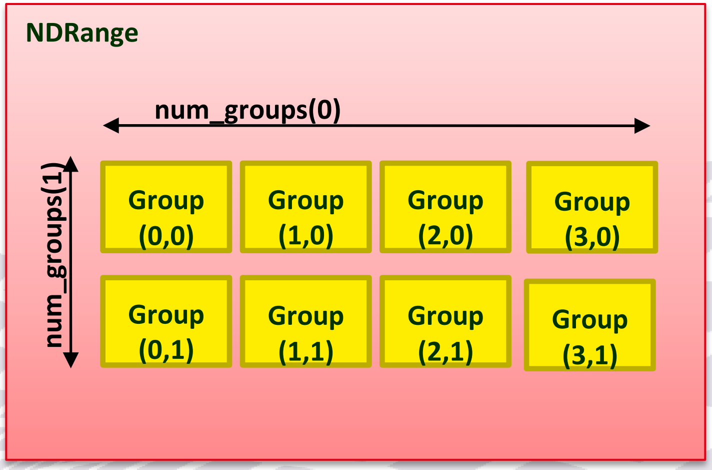
- Work items and work groups have IDs
  - So each thread can decide what data to work on
  - W.G. ID: ND
  - W.I. ID: ND
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - …



Work Group and Work Item built-in functions

- Work Item keywords
  - get_local_id(dim) returns the index inside the W.G.
  - get_global_id(dim) returns the index inside the NDRange
  - get_local_size(dim) returns the block size
  - get_global_size(dim) returns the NDRange size
- Work Group keywords
  - get_group_id(dim) returns the work group index
  - get_num_groups(dim) returns the grid size





OpenCL Application

OpenCL platform

- OpenCL platform = host + OpenCL devices
  - Devices can be CPUs, GPUs and so…
  - At the moment, Nvidia SDK does not implement OpenCL on CPUs
    - ATI allows to use OpenCL either on GPUs or x86 CPUs (what are AMD and Intel's)
- Available OpenCL devices depend on their drivers
  - Only the driver is going to tell OpenCL of the capabilities of its hardware
  - CPUs have no driver
- Drivers for Nvidia and ATI GPUs cannot be installed on a same host
  - So this is one platform or another
  - But take care, maybe one day it will be possible

The execution context

- Get the list of all platforms
  - Return the list of OpenCL driver installed on the system
  - clGetPlatformsIDs
  - clGetPlatformInfo

- Get the list of devices from a platform
  - Get list of available devices of a platform
  - clGetDeviceIDs
  - clGetDeviceInfo

- An execution context
  - Depends on a list of devices
  - clCreateContext

OpenCL Programs

- An OpenCL program consists of a set of kernels

- A program object contains
  - An associated context
  - A program source code
  - The kernel objects associated to

- Two ways to create a OpenCL program
  - From a string containing the kernel code
  - From a binary file (specific to an architecture)

- The "__kernel" keyword identifies a kernel

Creating Program & Kernel Objects

- The source program is a string
  - Contains kernels
  - Reads a file at runtime
  - Allows to modify kernels without recompiling the application
  - clCreateProgramWithSource

- Build the program object
  - Compile the program for a list of devices
  - clBuildProgram

- How to get kernels objects ?
  - Depends of program object
  - clCreateKernel

Kernel Syntax

- In the kernel, the language must be OpenCL C
- Built-in Functions
  - int get_local_id(int dim)
  - int get_global_id(int dim)
  - int get_local_size(int dim)
  - int get_global_size(int dim)
  - …
- Math functions
  - Trigonometric functions
  - Mad function
  - Min and max functions
  - Exponential, logarithm and power
  - modulus
  - …

The Command Queue

- An OpenCL command-queue
  - Used to queue a set of operations
  - Associated to a unique device
  - In-order or out-of-order
  - clCreateCommandQueue

- To synchronize a command queue
  - clFlush
  - clFinish

- Having multiple command-queues allows applications to queue multiple independent commands without requiring synchronization

Device Memory

- Two types of memory objects
  - Buffer Objects : 1D memory
  - Image Objects : 2D-3D memory

- Any data transfer to/from the device implies
  - A host pointer
  - A device memory object
  - The size of data (in bytes)
  - The command queue to enqueue
  - If it is a blocking transfer or not

- To transfer to/from the device
  - clEnqueueWriteBuffer
  - clEnqueueReadBuffer

Kernel launch

- Set kernel arguments
  - Cause of kernel compilation at runtime
  - clSetKernelArg

- 2 way to launch kernels
  - Enqueue task kernel
  - Enqueue NDRange kernel

- Set kernel options
  - Set global size of NDRange
  - Set local size of workgroups

- Kernel launches are asynchronous

OpenCL Kernel Performance

PRACE/LinkSCEEM Winter School January, 26th 2011

Agenda

- OpenCL Image Objects
- Local Memory
- Transformations & Optimizations
- Non-portability of the performances

OpenCL Image Objects

OpenCL Image Support

- Allows to allocate 2D or 3D arrays in a specific object
  - Allow to exploit some hardware functions
  - Based on the Textures in OpenGL

- Allows vector types, by using specific image format
  - The most common is float4, but float, float2 and various integer types are possible
  - List of available formats is implementation-dependant
  - 3D images are not required for Embedded Profile

- Elements of an image are stored in an opaque format
  - Cannot be directly accessed using a pointer

Images in kernel

- Memory object argument of a kernel needs a qualifier
  - __read_only (by default)
  - __write_only
  - Kernel cannot read from and write to the same image memory object
- Like with OpenGL textures, many options for the sampler
  - Normalized coordinates or not
  - Address can be clamped or in tiling mode
  - Results can be the nearest value, or a linear interpolation
- One of the many, many functions:

```
float4 read_imagef (image2d_t image,
                    sampler_t sampler,
                    int2 coord);
```

```
void write_imagef (image2d_t image,
                   int2 coord,
                   float4 color);
```

To use or not to use Images

- Images cannot be updated (read & write in the same kernel)
  - useless for instance for the “C” matrix in xGEMMs
- Therefore, they can be cached very effectively when reading : no cache coherency needed
  - In particular, images are very, very good on ATI 58xx
  - Not so much on Nvidia C2050, as global memory is also cached
- Images are not necessarily native to all hardware
  - Neither IBM Power 7 nor IBM Cell B/E support Images
- Can be extremely efficient if the linear interpolator can be leveraged
  - Unfortunately not a very common thing in HPC codes

Vectorization

- OpenCL specifications part 6.1.2 (types), 6.4 (operations)
- Vector data types (float4, …) represent a pseudo-array of elements (“components” in OpenCL)
- OpenCL specifies per-component semantic
    - So the product of two float4 is another float4, not a scalar float : it is not a dot product
    - Scalar are implicitly converted into the proper type by replicating the value
- Much more efficient on vectorized architecture, as only ¼ (for float) of the peak is reachable in scalar mode
- Component can be accessed in different ways
    - Scalar: temp.s0, temp.s1
    - Sub-vector: temp.odd, temp.left

Local Memory

Local Memory

- OpenCL specifications, part 3.3
- Local memory is local to a work-group
  - Shared by all work-items of work-group
  - Can be used for sharing between work-items
  - Can be used to cache global memory

- Low latency access
  - On Nvidia C1060-class hardware, “Nvidia shared memory” has the same latency as registers

- 2 ways to allocate it
  - Statically, inside the kernel
  - Dynamically, from the host as a parameter

Local Memory Usage

- Many work-items of a work-group have access to the same subset of data
  - Allows spatial locality between work-items
  - Allows temporal locality in one or more work-items
  - On Cell, is the only way to efficiently access data
- Can be used to realign memory
  - On Nvidia, avoid uncoalesced access in global memory by pre-loading aligned block into local memory
- To synchronize work-items of a work-group
  - Optional atomic operations in local memory (part 9.6)
- Be careful about bank conflict on Nvidia Hardware

Allocation

- Inside a kernel
  - Declare local memory as a static array
  - Use the keyword as a type prefix
    - __local or local

- Declare a Kernel with parameter in local memory
  - Allocation done during set of kernel arguments

Transformations & Optimizations

Memory Transfer Optimization

- 2 types of transfers
  - Blocking (“synchronous”)
  - Non-Blocking (“asynchronous”)

- In the function clEnqueueRead/Write, set the parameter blocking
  - CL_TRUE, make a blocking transfer
  - CL_FALSE, make a non-blocking transfer

- For a non-blocking transfer
  - Needed to control asynchronism
  - Allows to synchronize commands, command queues and the host

Memory Optimization

- On Nvidia GPU
  - Global memory access > 400-700 cycles
  - Prefers coalesced, small-size accesses (4 bytes per threads)
- On AMD/ATI GPU
  - Data reuse from Global (readings cached/writings not cached)
  - Images can be very fast
  - Prefers access to global in larger chunks (16 bytes, i.e. float4 per threads)
- Optimize the use of register
  - Depends of the number of blocks on Nvidia Hardware
- Use the local memory
  - Low latency access

NDRange Optimization

- Optimization of the NDRange is hardware dependant
- Required for efficient use of GPUs
  - Must combine the global & local size for maximum efficiency of the various cores & the memory subsystem

- On Nvidia GPU, WG size must be multiple of warp size
  - Like with CUDA try different size
    - 16x16
    - 32x4
    - 64x1
    - …
- If it works well in CUDA, then it should also works well in OpenCL :-)
  - Each generation of hardware has different requirements

NDRange Optimization

- On ATI GPU, WG size must be multiple of wavefront size
  - More WI can be generated
  - GPU can switch when a WI is stalled
  - Hiding memory latency
  - Optmizing execution

- On IBM Cell, the number of WG must be a multiple of the number of SPU
  - work-item of a work-group are merged (unrolled) in a single thread, executing on a single SPU
  - Large multiple are not necessarily better, having the exact number can be the most efficient choice

Code Optimizations & Transformations

- Loop Unrolling
  - Beware of rest loop
  - Fullunroll for small loops
  - Split space iteration (Nvidia Hardwares)

- Software pipelining
  - Increases the distance between a memory load and its use
  - hides part of the latency

- Avoid conditional
  - On GPU, avoid divergent branches
  - Use predication rather than control-flow when possible

Native functions

- Native functions are implementation defined
  - native_* prefix identifier

- Native functions are faster than built-in functions
  - But precision is implementation-dependant

- Available operation
  - Division
  - Arithmetic : cos, sin, tan
  - Logarithm : base-e, 2, 10
  - Exponential : base-e, 2, 10
  - Square root and inverse

Hardware dependant : Nvidia GPU

- Global memory optimization
  - Think about coalescing : alignment, granularity, size of accesses
  - Access to shared memory without banck conflicts
  - Ideal pattern has changed on Fermi, with 32 banks instead of 16
- NDRange optimization
  - Like CUDA grid
- Use native_* math functions where possible
- Partition the computation
  - nbWorkgroups >> nbMultiprocessor
  - Keep resources low enough ; the register file & the local memory are shared by all work-item in all work-groups executing simultaneously on a multiprocessor

Hardware dependant : ATI/AMD GPU

- Think about vectorization
  - float4 is best, in particular for global memory accesses
  - Helps in computation also
- NDRange optimization
  - 64 work-item per work-group
  - Multiple work-item per wavefronts
- Data reuse
  - Reads are cached
  - But not writes

Non-portability of the performances

The case of SGEMM

- Classic, well-known operation
  - Should be extended to DGEMM when double-precision support becomes available on all platforms…
- Quick study in OpenCL, comparing on ATI GPU, Nvidia GPU, IBM Cell B/E
  - The CPU is the more exotic of the three :-)
- Completely different code in each case !
- Kernel validated on GPUs by integration in HMPP codelet

SGEMM on ATI

- Kernel from the GATLAS http://golem5.org/gatlas/ OpenCL generator
    - Generates many forms of vectorized OpenCL SGEMM kernels, adapted to each problem size
- Non-vectorized kernel are very inefficient on ATI hardware
- 8x8 is the default, and the most efficient work-group size
- Grid size is large, and depend on internal unrolling in the generator
- Using images for input matrices A & B, is much, much better
- On n=m=k=4000 (4096 is noticeably less efficient)
    - About 500 Gflops with buffer on HD5870
    - About 1.3 Tflops with images !

SGEMM on NVidia

- Using GATLAS-derived kernels (current version of GATLAS doesn’t support non-vectorized kernels, work is in progress) or HMPP-generated kernels
- vectorized kernels are inefficient on C1060
  - Float4 memory accesses don’t coalesce very well, and generates lots of local memory bank conflicts
  - Execution units are not vectorized
- Non-vectorized kernels are good
  - Opposite reason !
- Compared to ATI, larger unrolling factor
  - Compensating for lack of vectorizations
- Images couldn’t be tested

SGEMM on Cell

- Best non-OpenCL matmult on Cell at http://www.tu-dresden.de/zih/cell/matmul
- Achieve >99% of peak under ideal conditions…
  - Assembly code, optimized DMA transfers, double buffering, …
- Current OpenCL Kernel only 65% of peak :-(
- Requires use of async_work_group_copy (not used on GPU) to use DMA for global -> local copies
- Requires massively unrolled float4 arithmetic
- Also partial double buffering, software pipelining to hide DMA latencies, …
- Much more similar to traditional CPU optimizations

Hands-on : Convolution

Hands-on : Convolution

- 6 incremental steps :
  - Communication part in function.cc
  - Naive Kernel
  - Kernel with static Local memory
  - Kernel with dynamic Local memory
  - Full unroll loops
  - Put hard coded coefficient

- Convolution :

$$
\begin{aligned}
B[i][j] &= 2 * Cst0 * A[i][j] \\
&+ Cst1 * (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) \\
&+ Cst2 * (A[i-2][j] + A[i+2][j] + A[i][j-2] + A[i][j+2]) \\
&+ Cst3 * (A[i-3][j] + A[i+3][j] + A[i][j-3] + A[i][j+3]) \\
&+ Cst4 * (A[i-4][j] + A[i+4][j] + A[i][j-4] + A[i][j+4])
\end{aligned}
$$

Vasnier Jean-charles

Application Engineer

jvasnier@caps-entreprise.com

+33 (0) 222 511 600