



OpenCL

General Purpose Computations on GPU

Martin Kruliš

Outline

- ▶ GPGPU History
- ▶ Current GPU Architecture
- ▶ OpenCL Framework
- ▶ Example (and its Optimization)
- ▶ Alternative Frameworks
- ▶ Most Recent Innovations

History

1996: 3Dfx Voodoo 1

- First graphical (3D) accelerator for desktop PCs

1999: NVIDIA GeForce 256

- First Transform&Lightning unit

2000: NVIDIA GeForce2, ATI Radeon

2001: GPU has programmable parts

- DirectX – vertex and fragment shaders (v1.0)

2006: OpenGL, DirectX 10, Windows Vista

- Unified shader architecture in HW
- Geometry shader added

History

2007: NVIDIA CUDA

- First GPGPU solution, restricted to NVIDIA GPUs

2007: AMD Stream SDK (previously CTM)

2009: OpenCL, Direct Compute

- Mac OS (Snow Leopard) first to implement OpenCL

2010:

- OpenCL implementation from AMD and NVIDIA
- OpenCL revision 1.1

2011: OpenCL 1.2 (current stable)

2012: NVIDIA Kepler Architecture

GPU in comparison with CPU

▶ CPU



- Few cores per chip
- General purpose cores
- Processing different threads
- Huge caches to reduce memory latency
 - Locality of reference problem

▶ GPU



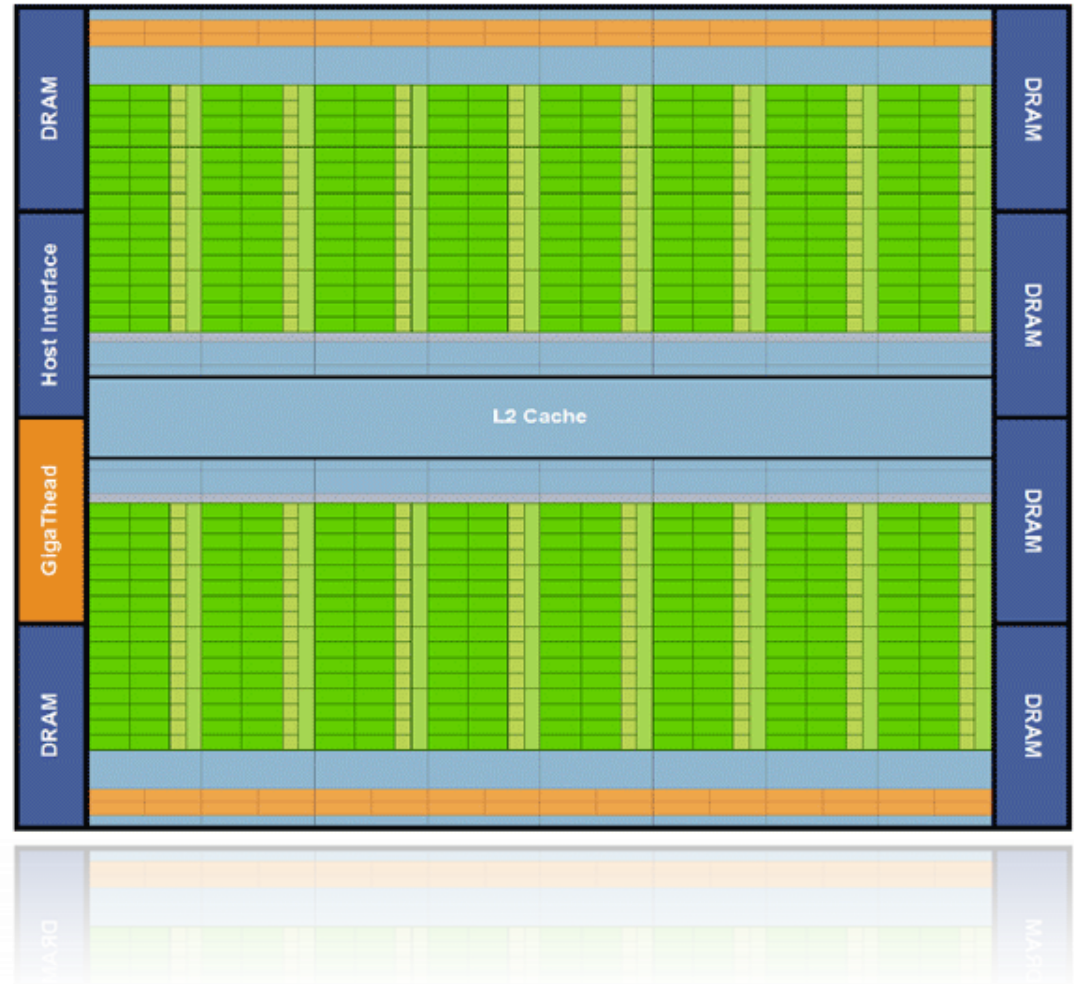
- Many cores per chip
- Cores specialized for numeric computations
- SIMT thread processing
- Huge amount of threads and fast context switch
 - Results in more complex memory transfers

Current GPU Hardware

▶ NVIDIA Fermi

- 16 SMP units
- 512 CUDA cores
- 786kB L2 cache

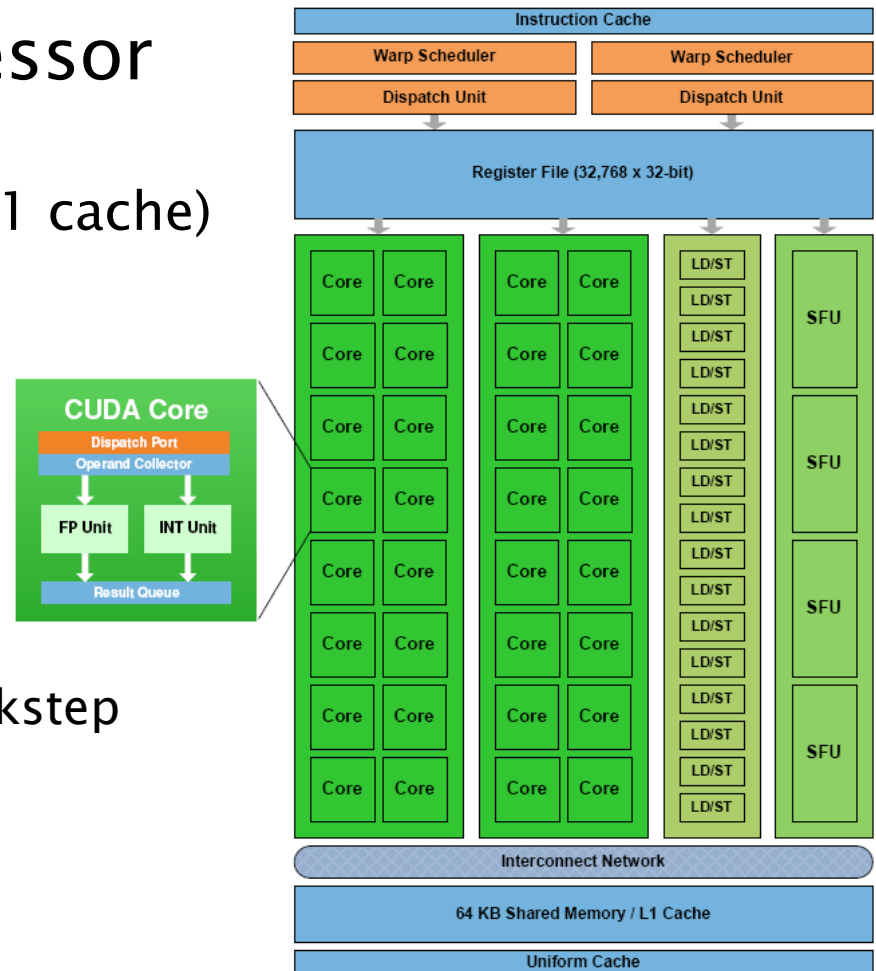
Note that one CUDA core corresponds to one 5D AMD Stream Processor (VLIW5). Therefore Radeon 5870 has 320 cores with 4-way SIMD capabilities and one SFU.



Current GPU Hardware

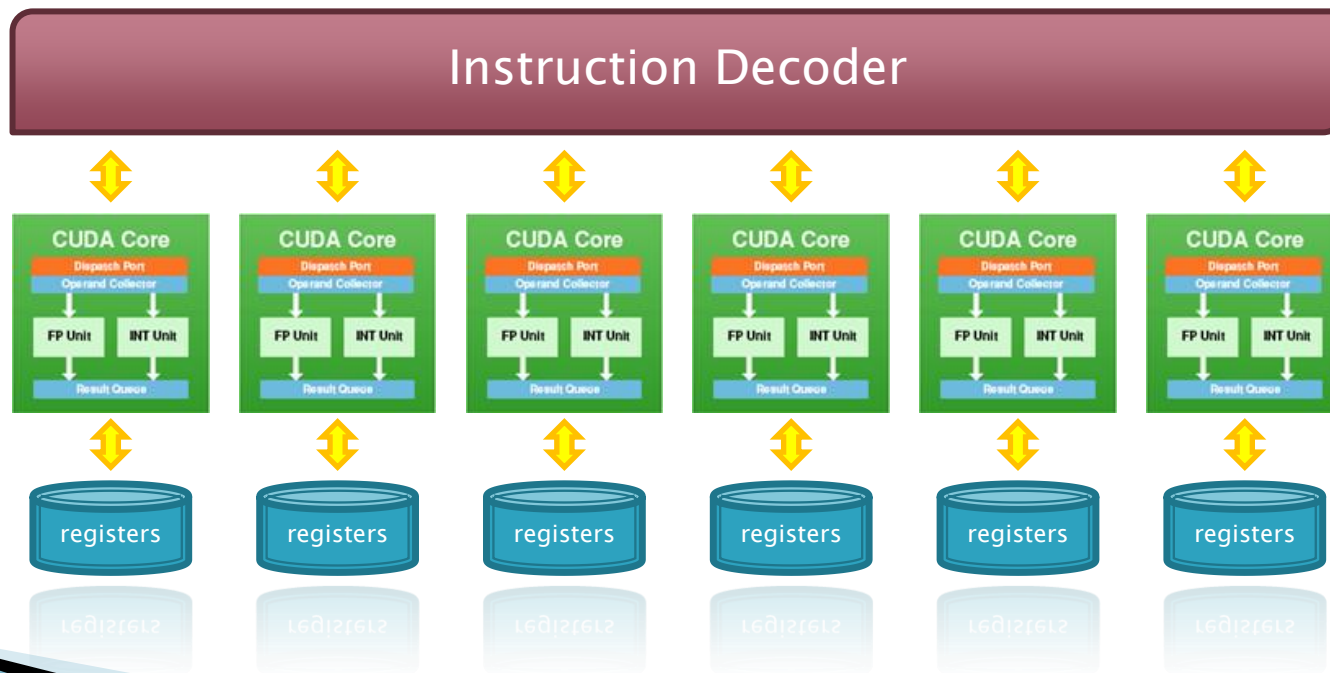
▶ Streaming Multiprocessor

- 32 CUDA cores
- 64kB shared memory (or L1 cache)
- 1024 registers per core
- 16 load/store units
- 4 special function units
- 16 double precision ops per clock
- 1 instruction decoder
 - All cores are running in lockstep



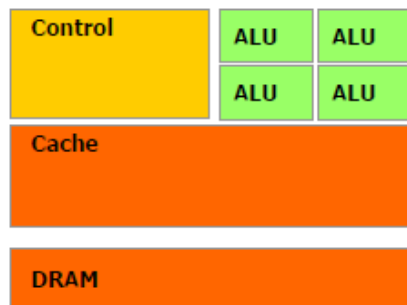
SIMT Execution

- ▶ Single Instruction Multiple Threads
 - All cores are executing the same instruction
 - Each core has its own set of registers

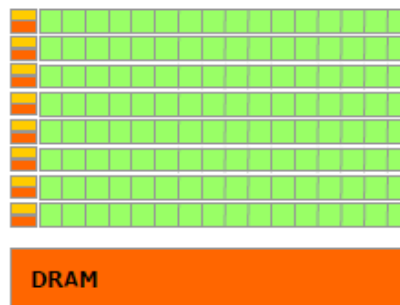


Dealing With Memory Latency

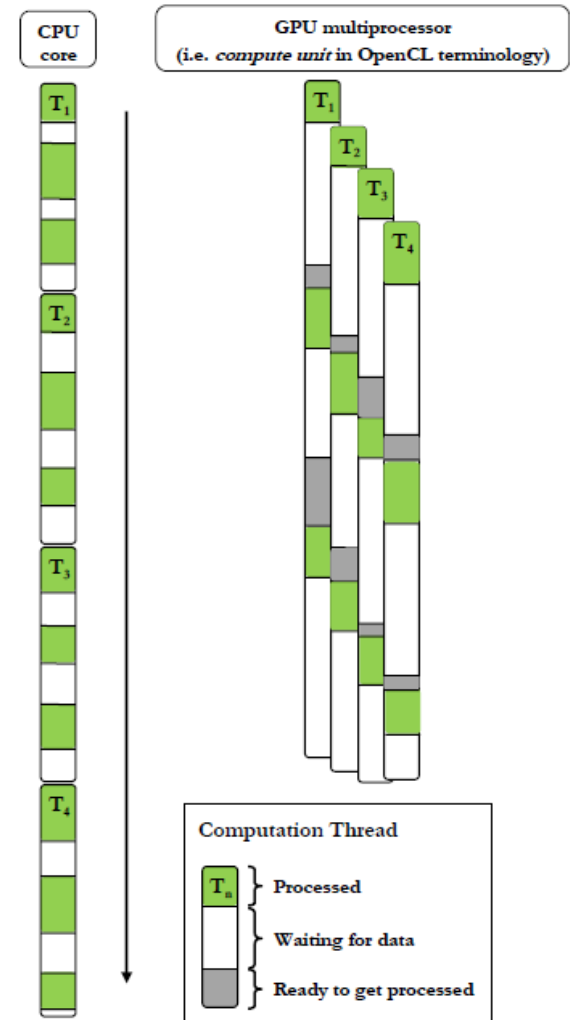
- ▶ CPU
 - Expensive context-switch
 - Large caches required
- ▶ GPU
 - Fast context switch
 - Another thread (warp) may run while current is stalled
 - Small caches



CPU



GPU



OpenCL

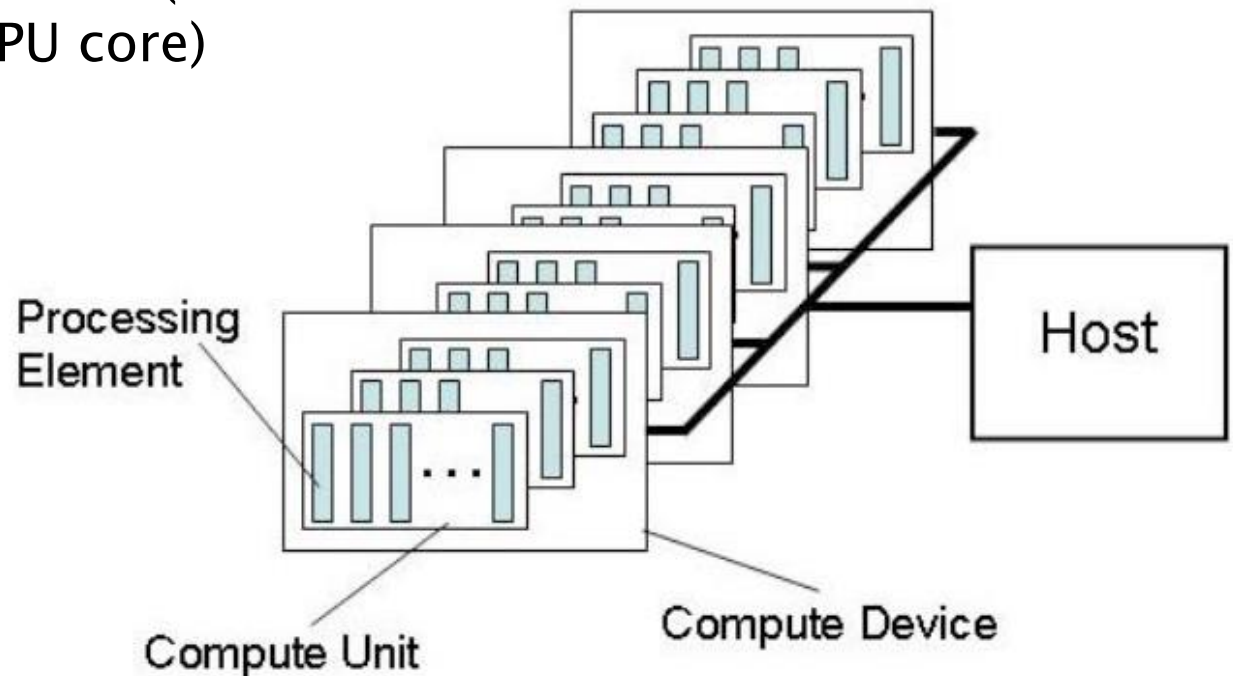


OpenCL

- ▶ **Universal Framework for Parallel Computations**
 - Specification created by Khronos group
 - Multiple implementations exist (AMD, NVIDIA, Mac, ...)
- ▶ **API for Different Parallel Architectures**
 - Multi-Core CPU, Many-Core GPU, IBM Cell cards, ...
 - Handles device detection, data transfers, and code execution
- ▶ **Extended Version of C99 for Programming Devices**
 - The code is compiled at runtime for selected device
 - Theoretically, we may chose best device for our application dynamically
 - However, we have to consider HW-specific optimizations...

OpenCL – Hardware Model

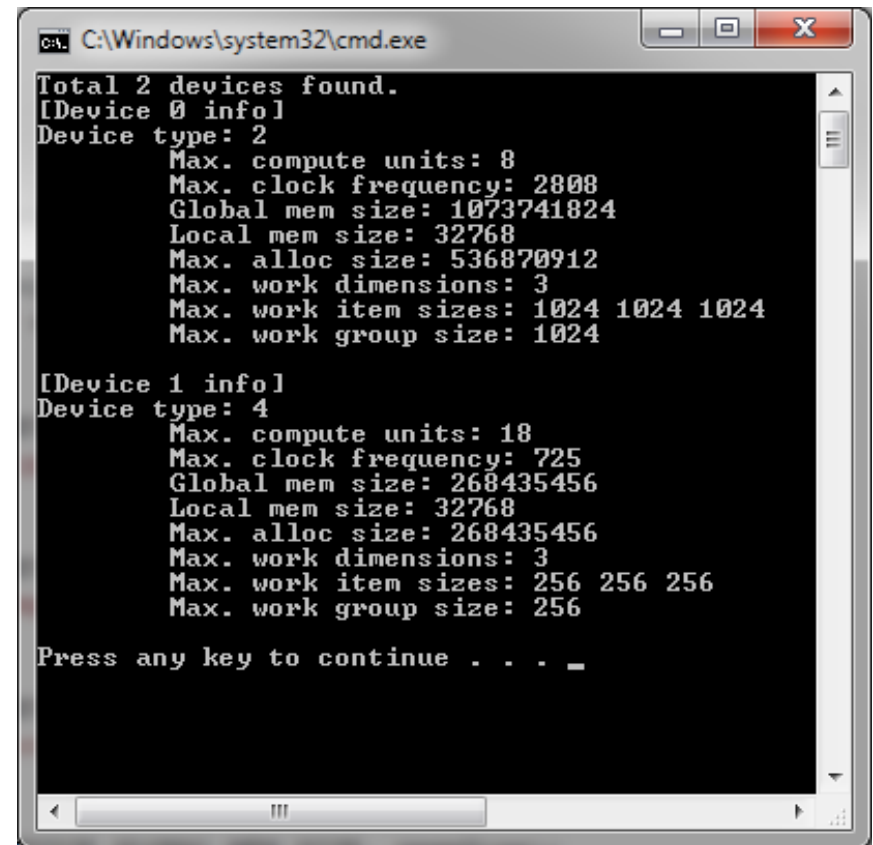
- ▶ Hardware is mapped to the following model...
 - Device (CPU die or GPU card)
 - Compute unit (CPU core or GPU SMP)
 - Processing element (slot in SSE registers or GPU core)



OpenCL – Logical Model

▶ Logical Layers

- Platform
 - An implementation of OCL
- Context
 - Groups devices of selected kind
 - Buffers, programs, and other objects lives in context
- Device
- Command Queue
 - Created for a device
 - One device may have multiple command queues



```
C:\Windows\system32\cmd.exe
Total 2 devices found.
[Device 0 info]
Device type: 2
Max. compute units: 8
Max. clock frequency: 2808
Global mem size: 1073741824
Local mem size: 32768
Max. alloc size: 536870912
Max. work dimensions: 3
Max. work item sizes: 1024 1024 1024
Max. work group size: 1024

[Device 1 info]
Device type: 4
Max. compute units: 18
Max. clock frequency: 725
Global mem size: 268435456
Local mem size: 32768
Max. alloc size: 268435456
Max. work dimensions: 3
Max. work item sizes: 256 256 256
Max. work group size: 256

Press any key to continue . . . _
```

Intel Core i7 (4 cores with HT)
ATI Radeon 5870 (320 cores)

OpenCL – Client Application

```
std::vector<cl::Platform> platforms;  
cl_int err = cl::Platform::get(&platforms);  
if (err != CL_SUCCESS) return 1;
```

List all platforms

```
cl_context_properties cps[3] = {CL_CONTEXT_PLATFORM,  
    (cl_context_properties) platforms[0](), 0};  
cl::Context context(CL_DEVICE_TYPE_GPU, cps, NULL, NULL, &err);
```

Context of all GPUs on
the 1st platform

```
std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
```

List all GPUs

```
cl::Buffer buf(context, CL_MEM_READ_ONLY, sizeof(cl_float)*n);
```

```
cl::Program program(context, cl::Program::Sources(1,  
    std::make_pair(source.c_str(), source.length())));  
err = program.build(devices);
```

Create and compile the
program from string source

```
cl::Kernel kernel(program, "function_name", &err);  
err = kernel.setArg(0, buf);
```

Mark function as a kernel

```
cl::CommandQueue commandQueue(context, devices[0], 0, &err);  
commandQueue.enqueueWriteBuffer(buf, CL_TRUE, 0, sizeof(cl_float)*n, data);  
commandQueue.enqueueNDRangeKernel(kernel, cl::NullRange, cl::NDRange(n), cl::NDRange(grp),  
    NULL, NULL);  
commandQueue.finish();
```

GPU command queue

Send commands and wait for them to finish

OpenCL – Kernels

▶ A Kernel

- Written in OpenCL C (extended version of C99)
- Compiled at runtime for destination platform
 - With high degree of optimization

▶ Kernel Execution

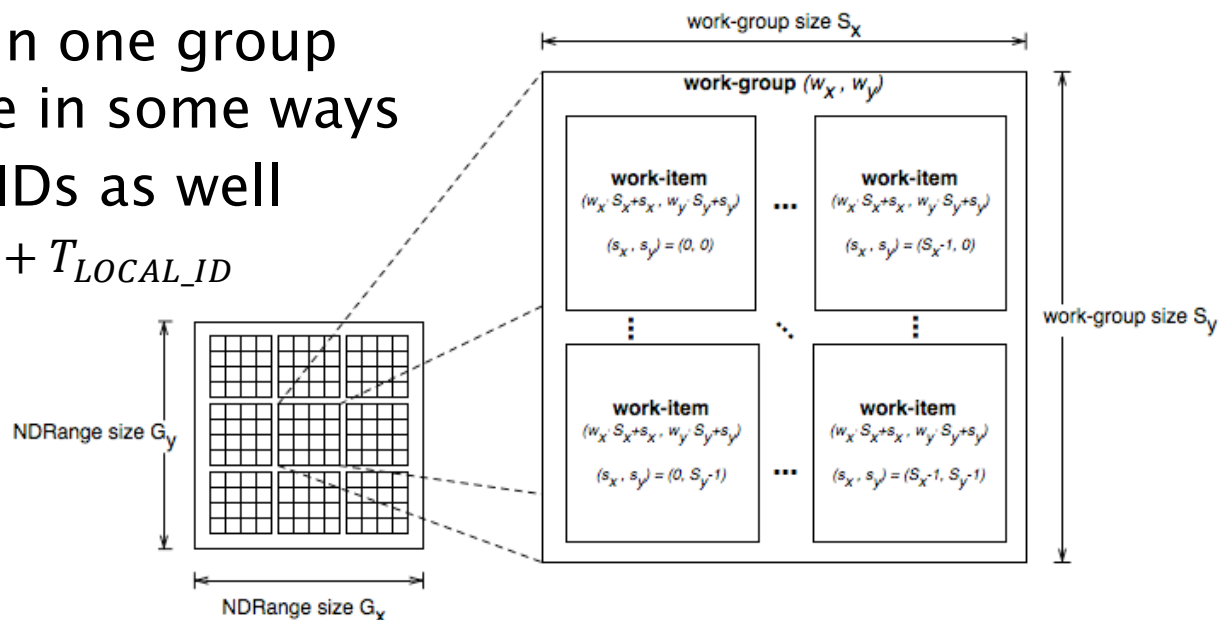
- Task Parallelism
 - Multiple kernels are enlisted in command queue and executed concurrently
- Data Parallelism
 - Multiple instances (threads) are created from single kernel, each operate on distinct data

OpenCL – Kernel Execution

▶ Data Parallelism

- Each kernel instance has its own ID
 - A 1–3 dimensional vector of numbers from 0 to N–1
- ID identifies the portion of data to be processed
- Threads form groups
 - Threads within one group can cooperate in some ways
 - Groups have IDs as well

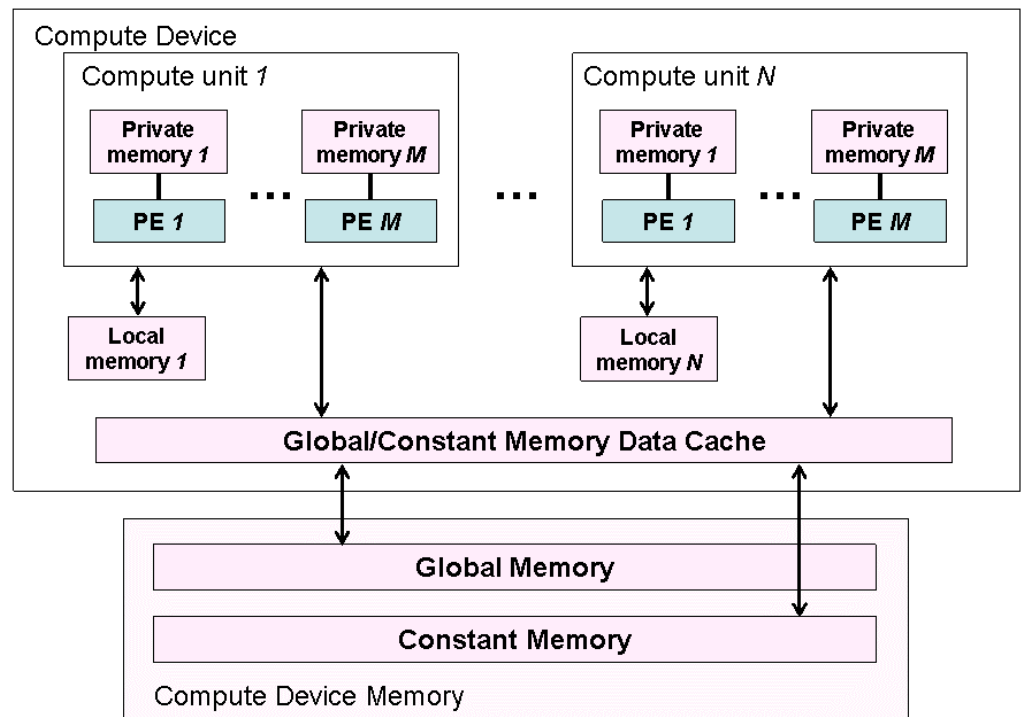
$$T_{ID} = G_{ID}G_{SIZE} + T_{LOCAL_ID}$$



OpenCL – Memory Model

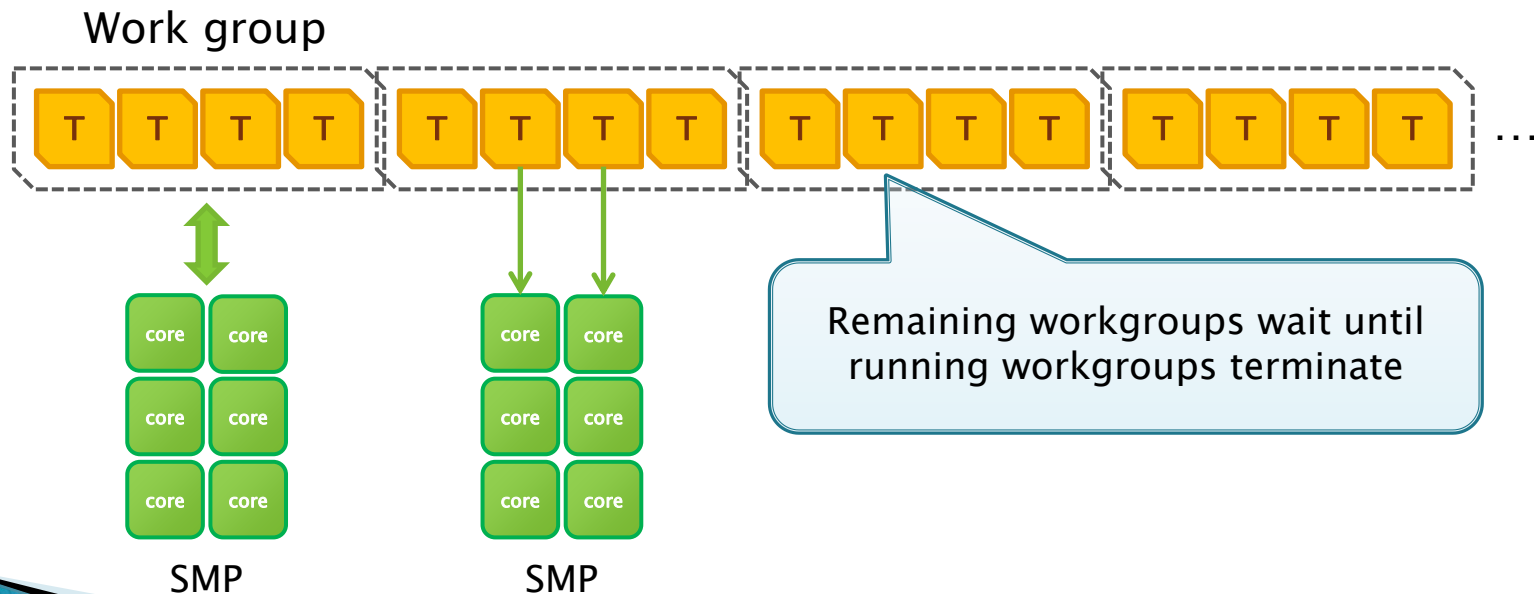
► Types of Memory

- **private** – memory that belongs to one thread
- **local** – memory shared by workgroup
- **global** – memory of the device
- **constant** – read-only version of global memory



How It Really Works (on GPU)

- ▶ Threads and work groups mapping to hardware
 - Workgroup is assigned non-preemptively to SMPs
 - Threads are mapped to cores
 - Multiple threads may be mapped to one core



OpenCL – Programming Kernels

▶ Data Types

- Almost every standard C99 types (`int`, `float`, ...)
- Some predefined types: `size_t`, `ptrdiff_t`
- Special type `half` – 16bit equivalent of `float`

▶ Vector Data Types

- `typeN`, where `type` is std. type and $N \in \{2,4,8,16\}$
 - E.g. `float4`, `int16`
- Items are accessed as structure members

```
int4 vec = (int4) (1, 2, 3, 4);  
int x = vec.x;
```

- Special swizzling operations are allowed

```
int4 vec2 = vec.xxyy;  vec = vec.wzyx;
```

OpenCL – Programming Kernels

▶ Functions

- Other functions (beside the kernel) can be defined in the program (kernel is just an entry point)
 - Calls are inlined on GPU, since there is no stack
- It is possible to call std. functions like `printf()`
 - However, it will work only on CPU
- There are many built-in functions
 - Thread-related functions (e.g. `get_global_id()`)
 - Mathematical and geometric functions
 - Originally designed for graphics
 - Some of them are translated into single instruction
 - Functions for asynchronous memory transfers

OpenCL – Programming Kernels

▶ Restrictions And Optimization Issues

- Branching problem (if–else)
 - Workgroup runs in SIMT, thus all branches are followed
 - Use conditional assignment rather than branches
- For–cycles
 - The compiler attempts to unwrap them automatically
- While–cycles
 - The same problem as branching
- Vector operations
 - Translated into single instruction if possible (e.g., SSE)
 - The compiler attempts to generate them automatically
 - Different efficiency on different architectures

OpenCL – Synchronization

▶ Global

- Explicit barriers added to the command queue
- Command queue operations event dependencies

▶ Within Workgroup

- Local barriers
- Memory fences
- Atomic operations (on integers)
 - For both local and global memory
 - Base and extended version
 - Base – common operations (add, sub, xchg, cmpxhg, ...)
 - Extended – min, max, and, or, xor

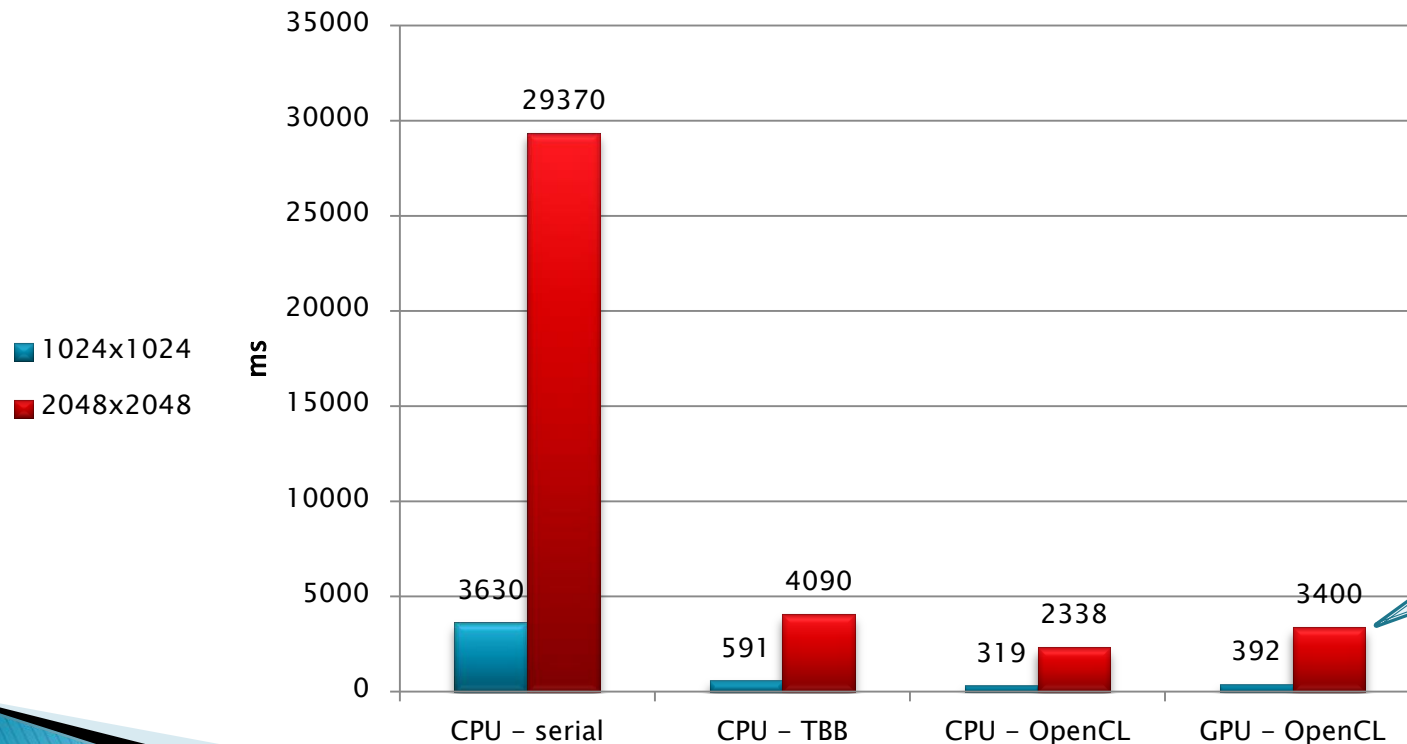
Example – Matrix Multiplication

```
__kernel void mul_matrix (__global const float *m1,  
    __global const float *m2, __global float *mRes)  
{  
    int n = get_global_size(0);  
    int r = get_global_id(0);  
    int c = get_global_id(1);  
    float sum = 0;  
    for (int i = 0; i < n; ++i)  
        sum += m1[r*n + i] * m2[c*n + i];  
    mRes[r*n + c] = sum;  
}
```

The second matrix is already transposed

Matrix Multiplication – Results

- ▶ Multiplication of two rectangular matrices
 - AMD Radeon 5870 (320 cores) vs. Core i7 (4 HT cores)
 - Naïve N^3 algorithm, second matrix is transposed



Looking for the cause...

► Profiler Results

How many times each thread read the global memory

Method	ExecutionOrder	GlobalWorkSize	GroupWorkSize	KernelTime	LocalMem	MemTransferSize	ALU	Fetch	Write
BufHostToDevice	1					4194304			
BufHostToDevice	2					4194304			
mul_matrix_Cypress	3	{1024; 1024; 1}	{16; 16; 1}	372,05272	0		5133	2048	1
BufDeviceToHost	4					4194304			

How many % of the time took memory reading ops

Wavefront	ALUBusy	ALUFetchRatio	ALUPacking	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	ALUStalledByLDS	LDSBankConflict
16384	6,34	2,51	35,99	94,37	83,15	0	0	0

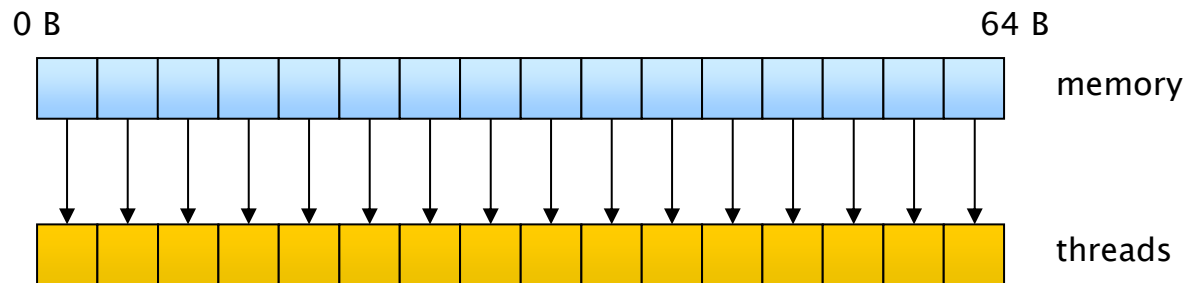
ALU to reading ops ratio

How many % of the time was fetch units stalled (waiting)

Global Memory Transactions

▶ Coalesced Load

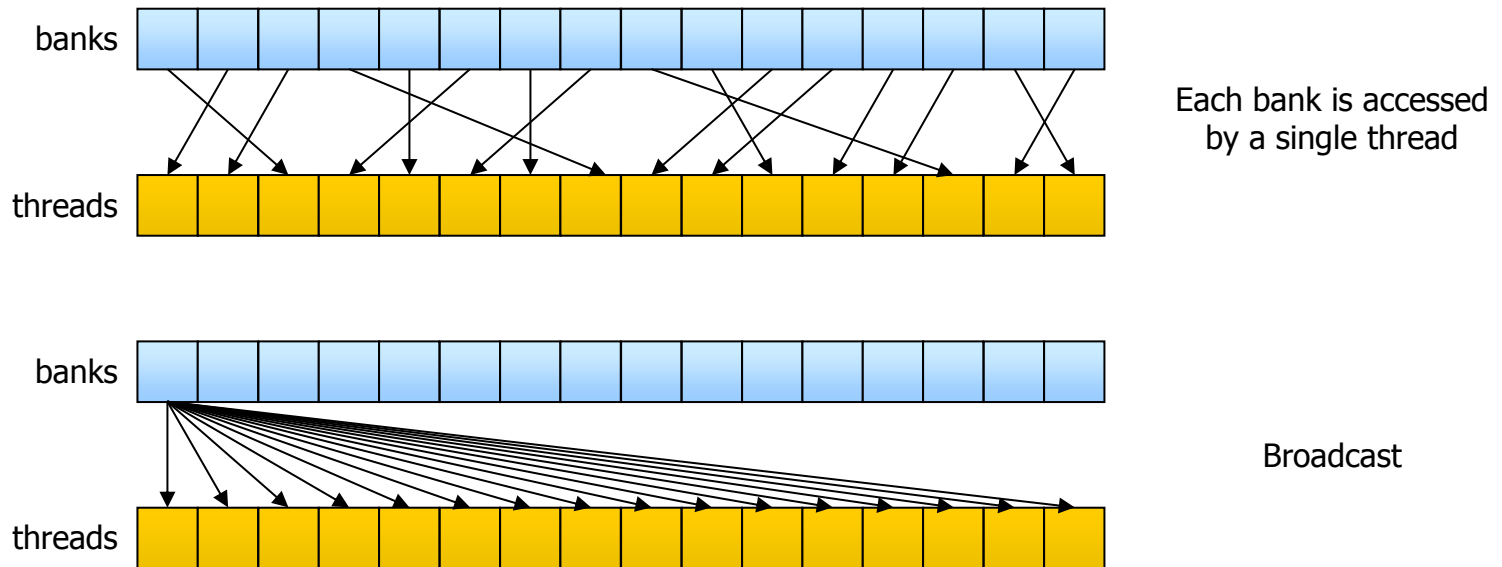
- Threads running in SIMT mode have to cooperate
- Each thread loads different 4-byte word from aligned continuous block
 - Details and rules for coalesced loads are different for each generation of GPUs
- HW performs such load as one memory transaction



Local Memory Model

▶ Banking

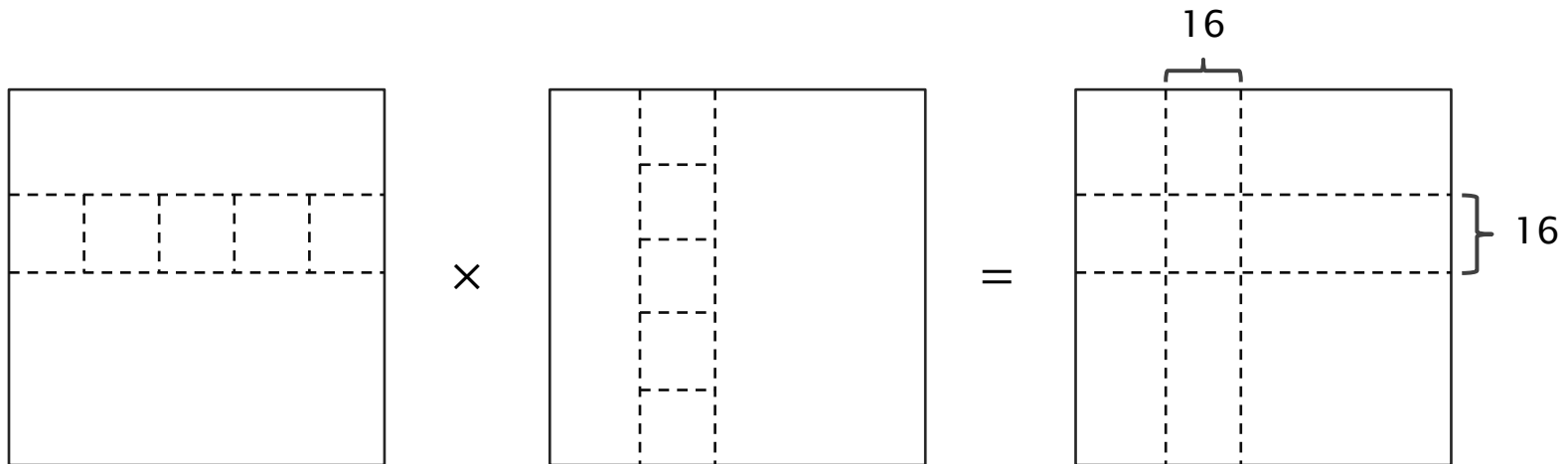
- Local memory is divided into banks
- Bank manages 4-byte words, addresses are assigned modulo number of banks (16 or 32)



Optimized Example

▶ Optimized Solution

- Workgroup computes block of 16x16 results
- In each step, appropriate blocks of 16x16 numbers are loaded into local memory and intermediate results are updated



Optimized Example

```
__kernel void mul_matrix_opt (__global const float *m1, __global const float *m2, __global float *mRes,
    __local float *tmp1, __local float *tmp2)
{
    int size = get_global_size(0);
    int lsize_x = get_local_size(0);
    int lsize_y = get_local_size(1);
    int block_size = lsize_x * lsize_y;
    int gid_x = get_global_id(0);
    int gid_y = get_global_id(1);
    int lid_x = get_local_id(0);
    int lid_y = get_local_id(1);
    int offset = lid_y*lsize_x + lid_x;

    float sum = 0;
    for (int i = 0; i < size; i += lsize_x) {
        tmp1[offset] = m1[gid_y*size + i + lid_x];
        for (int j = 0; j < lsize_x / lsize_y; ++j)
            tmp2[offset + j*block_size] = m2[(gid_x + lsize_y*j)*size + i + lid_x];
        barrier(CLK_LOCAL_MEM_FENCE);

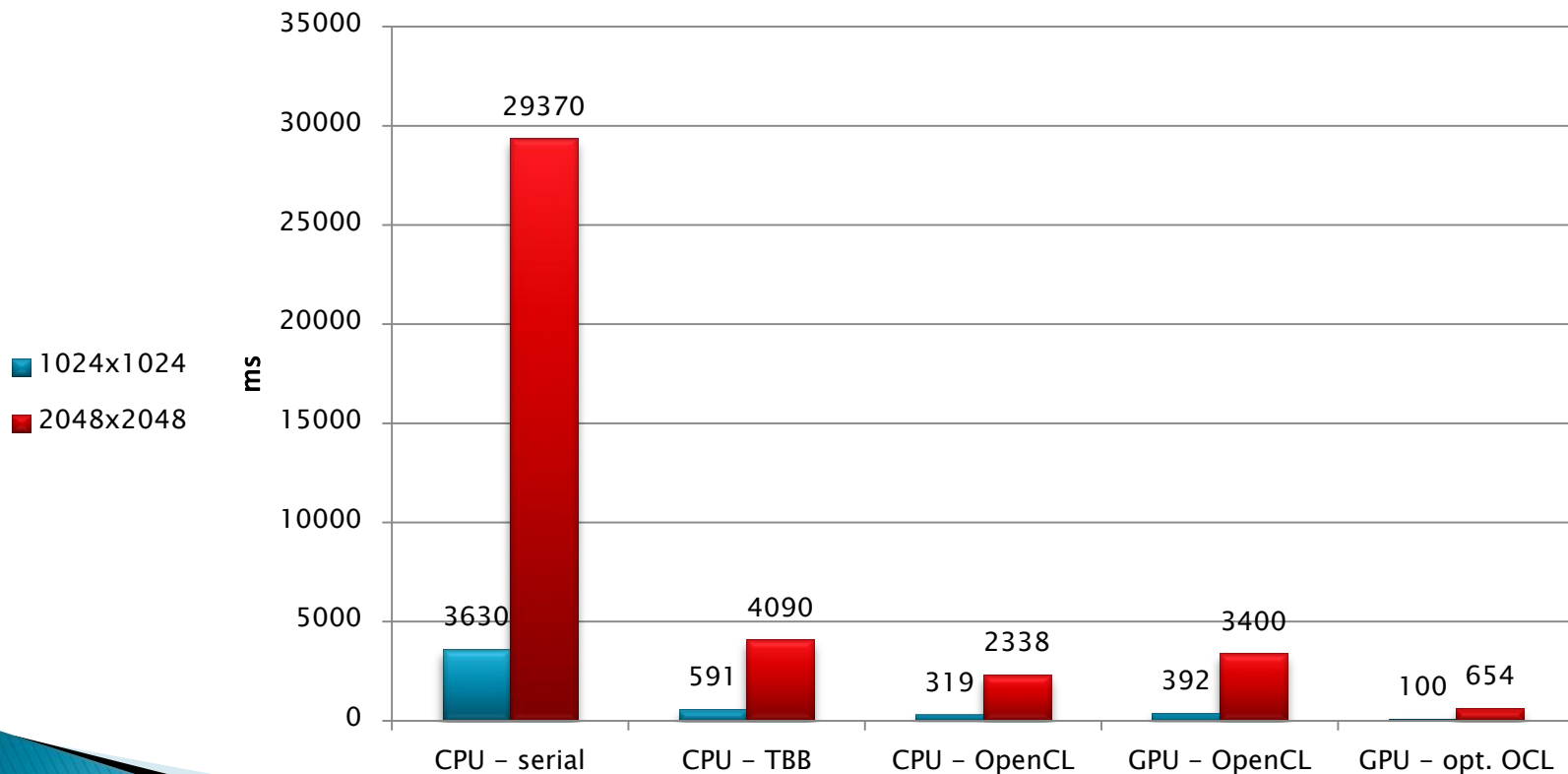
        for (int k = 0; k < lsize_x; ++k)
            sum += tmp1[lid_y*lsize_x + k] * tmp2[lid_x*lsize_x + k];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    mRes[gid_y*size + gid_x] = sum;
}
```

Copy corresponding blocks to local memory

Compute intermediate results from current blocks

Optimized Example Results

- ▶ Optimized version for GPU
 - 45x faster to serial CPU, 3.6x to parallel CPU version



Optimized Example Results

▶ Profiler Results

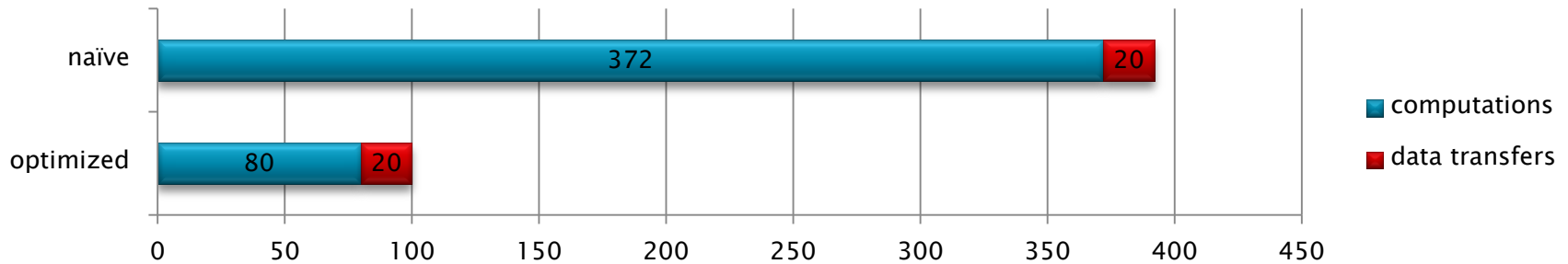
- Fetch has reduced from 2048 to 128 (16x)
- ALU operations took 45% of total time
- Data loads took only 9% and only 5% of time were fetch units stalled
- However, there were many bank conflicts

ALU	Fetch	ALUBusy	ALUFetchRatio	ALUPacking	FetchUnitBusy	FetchUnitStalled	WriteUnitStalled	ALUStalledByLDS	LDSBankConflict
7328	128	45,87	57,25	32,16	9,28	5,71	0	100	100

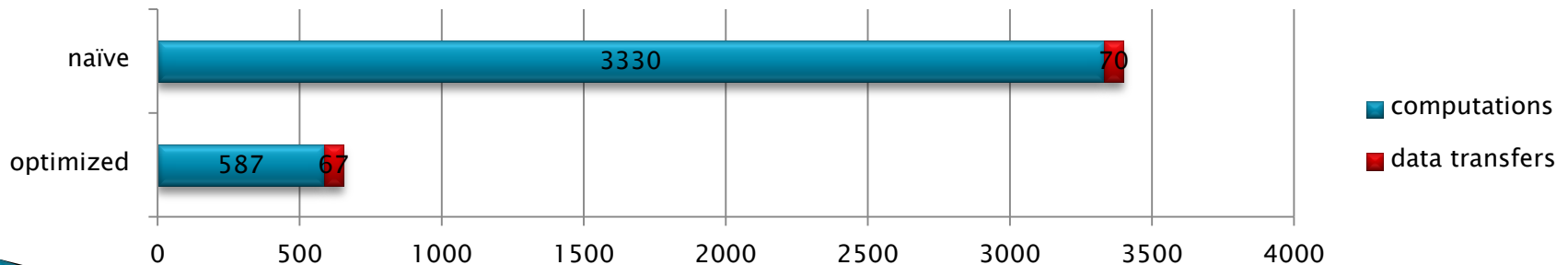
Data Transfers

▶ Data Transfers to Computations Ratio

- 1024x1024 matrix (8 MB to GPU, 4 MB from GPU)

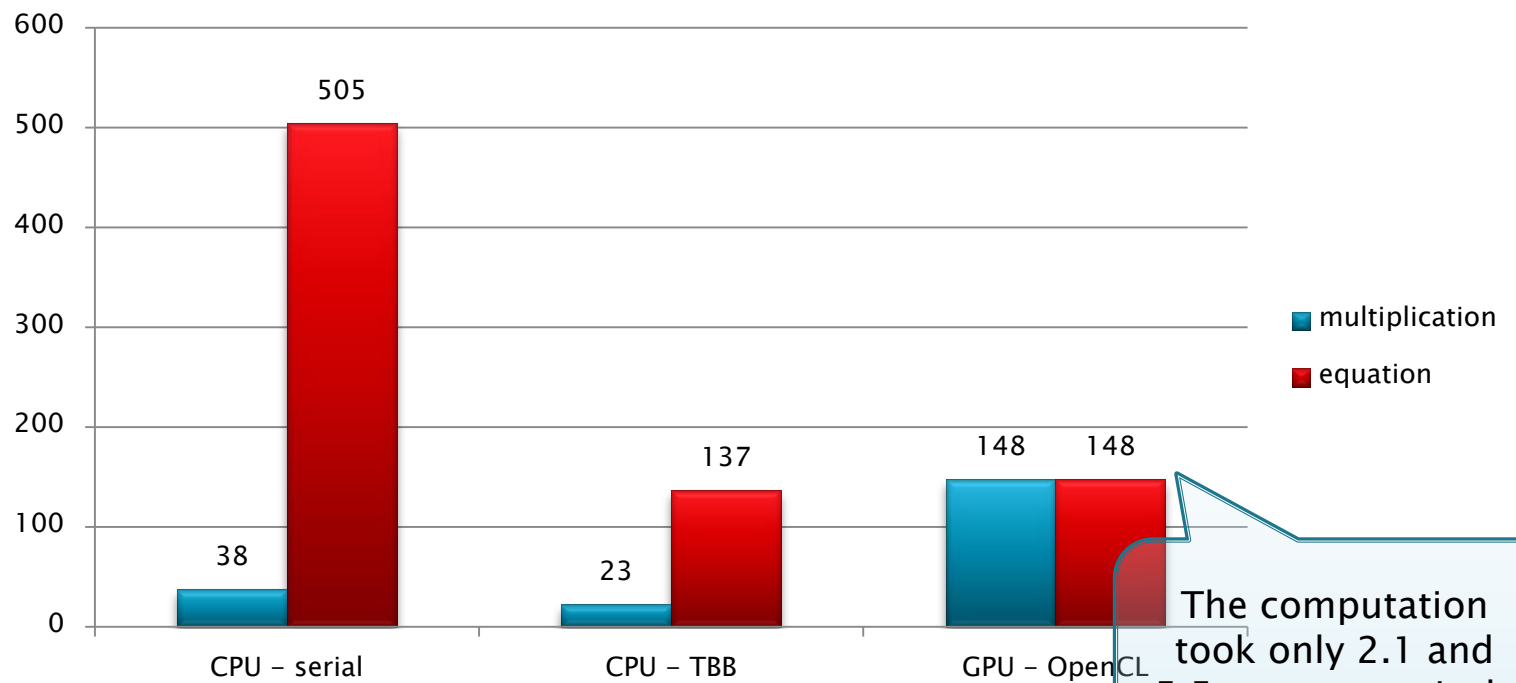


- 2048x2048 matrix (32 MB to GPU, 16 MB from GPU)



Other Examples – Vector Ops

- ▶ Two vectors of 16M floats
 - Multiplication: $z_i = x_i * y_i$
 - Equation: $z_i = \sqrt{x_i} * y_i/x_i + \cos y_i * x_i$

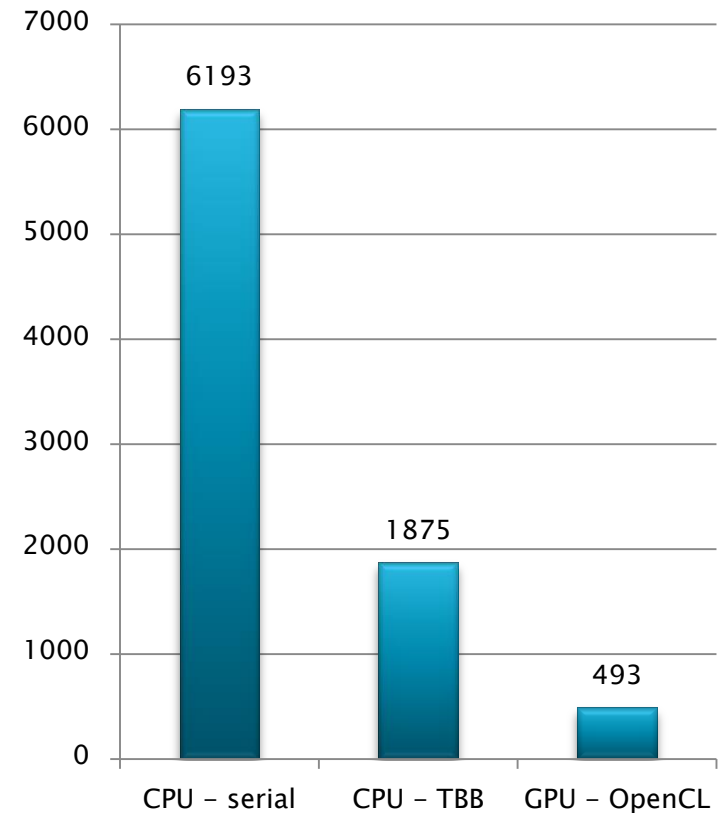


The computation took only 2.1 and 5.5 ms, respectively

Other Examples – Backtracking

▶ Special Version of Knapsack Problem

- Set of (30) numbers, each is added into the sum as either positive or negative
- Trying to find given sum
- Data modified, so the solution does not exist
 - There are only even numbers in the set and we are looking for odd one



Caveats And Pit-holes

- ▶ Drivers
 - Sometimes unstable, may cause OS crash
- ▶ Data
 - Needs to be transferred from host memory to GPU and back
- ▶ Task Parallelism on GPU
 - Currently only on NVIDIA Fermi and Kepler
- ▶ Kernel Compilation
 - Takes up to a few seconds

OpenCL And OpenGL

- ▶ OpenCL And OpenGL Relations
 - OpenCL is a younger brother of OpenGL
 - OpenCL has data types for representing images
 - And special types for representing colors
 - Many object conversions are defined
 - CL buffer to GL buffer
 - CL image object to GL texture
 - CL buffer to GL renderbuffer
 - OpenCL and OpenGL may share context
 - To create OpenCL objects from OpenGL objects

Alternative Technologies

▶ NVIDIA CUDA

- The first GPGPU technology
- More simple API, designed for GPUs only
 - No platform and device detection required
- Kernels are written directly into the main program

▶ Microsoft Direct Compute

- Part of DirectX 11 (from November 2009)
- Designed primarily for game developers
- API similar to vertex or fragment shaders

What's Up

- ▶ NVIDIA Kepler Architecture (CUDA 5.0)
 - Streaming Processors Next Generation (SMX)
 - 192 cores, 32 SFUs, 32 load/store units
 - 3 cores share a DP unit, 6 cores share LD and SFU
 - Dynamic Parallelism
 - Kernel may spawn child kernels (to depth of 24)
 - Implies the work group context-switch capability
 - Hyper-Q
 - Up to 32 simultaneous GPU-host connections
 - Better throughput if multiple processes/threads use the GPU (concurrent connections are managed in HW)

What's Up

- ▶ AMD's Graphic Core Next (GCN)
 - Abandoning the VLIW4 architecture
 - $1 \text{ VLIW} \times 4 \text{ ALU ops} \Rightarrow 4 \text{ SIMD} \times 1 \text{ ALU op}$
 - 32 compute units (Radeon HD7970)
 - 4 SIMD units per CU (each processing 16 elements)
 - 10 planned wavefronts per SIMD unit
 - Emphasis on vector processing (instructions, registers, memory, ...)
 - OpenCL 1.2, DirectCompute 11.1 and C++ AMP compatibility

Discussion

