

GPU Computing Tools

Varun Sampath

University of Pennsylvania

CIS 565 - Spring 2012

Agenda

- CUDA Toolchain
 - APIs
 - Language bindings
 - Libraries
 - Visual Profiler
 - Parallel Nsight
- OpenCL
- C++ AMP

CUDA Documentation

- <http://developer.nvidia.com/nvidia-gpu-computing-documentation>
- CUDA C Programming Guide
- CUDA C Best Practices Guide
- CUDA API Reference Manual
- Occupancy Calculator
- Much more

CUDA Organization

- Host code: two layers
 - Runtime API
 - `cudart` dynamic library
 - `cuda_runtime_api.h` (C)
 - `cuda_runtime.h` (C++)
 - Driver API
 - `nvcuda` dynamic library
 - `cuda.h`
- Device code
 - Kernel → PTX (parallel thread eXecution)

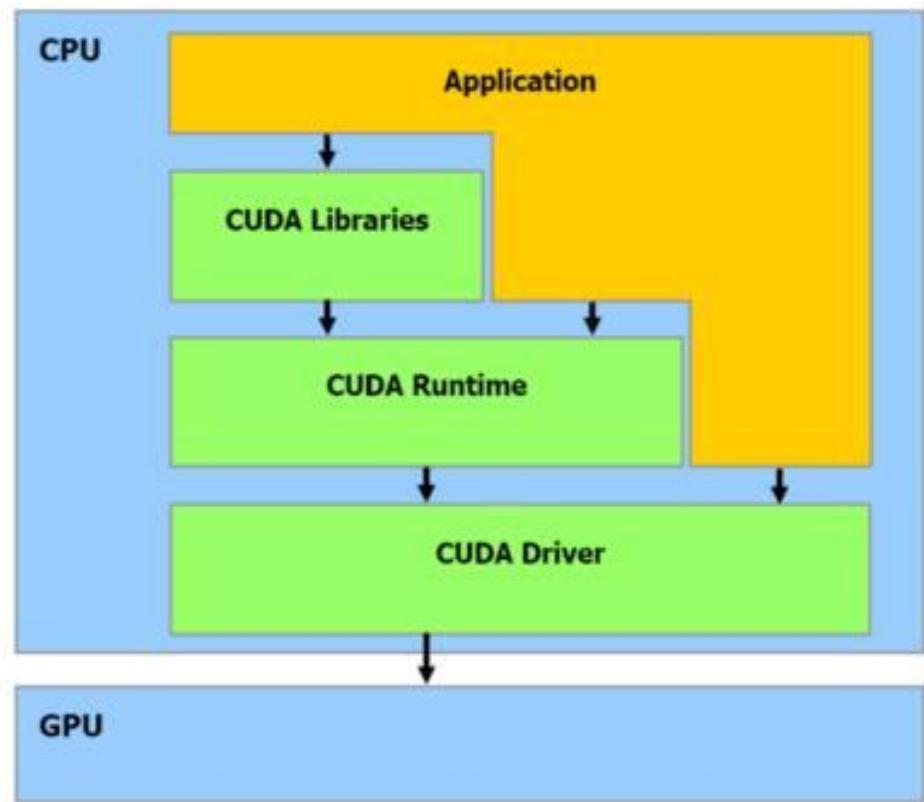


Image from [Stack Overflow](#)

CUDA API Comparison

CUDA Runtime API

```
// create CUDA device &  
context  
  
cudaSetDevice( 0 ); //  
pick first device  
  
kernel_naive_copy<<<cnBlo  
cks, cnBlockSize>>>  
(i_data, o_data,  
rows, cols);
```

CUDA Driver API

```
cuInit(0);  
cuDeviceGet(&hContext, 0);  
// pick first device  
cuCtxCreate(&hContext, 0,  
hDevice));  
  
cuModuleLoad(&hModule,  
"copy_kernel.cubin");  
cuModuleGetFunction(&hFuncti  
on, hModule,  
"kernel_naive_copy");  
  
...  
cuLaunchGrid(cuFunction,  
cnBlocks, 1);
```

Differences?

Some CUDA Language Bindings

- Note: the following are not supported by NVIDIA
- PyCUDA (Python)
 - Developed by Andreas Klöckner
 - Built on top of CUDA Driver API
 - Also: PyOpenCL
- JCuda (Java)
- MATLAB
 - Parallel Computing Toolbox
 - AccelerEyes Jacket

Whetting your appetite

```
1 import pycuda.driver as cuda
2 import pycuda.autoinit, pycuda.compiler
3 import numpy
4
5 a = numpy.random.randn(4,4).astype(numpy.float32)
6 a_gpu = cuda.mem_alloc(a.nbytes)
7 cuda.memcpy_htod(a_gpu, a)
```



[This is examples/demo.py in the PyCUDA distribution.]

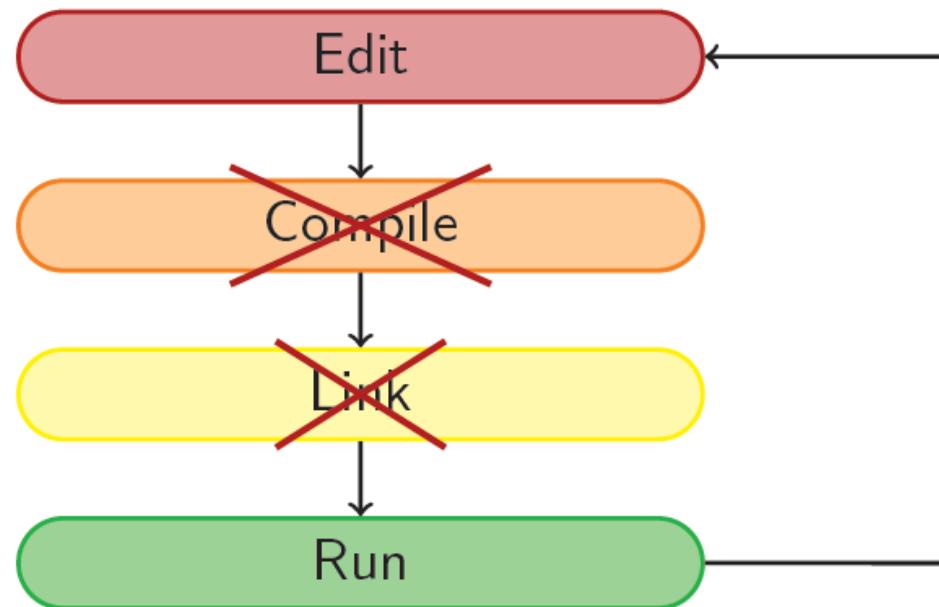
Whetting your appetite

```
1 mod = pycuda.compiler.SourceModule("""
2     __global__ void twice( float *a)
3     {
4         int idx = threadIdx.x + threadIdx.y*4;
5         a[ idx ] *= 2;
6     }
7     """
8 )
9 func = mod.get_function("twice")
10 func(a_gpu, block=(4,4,1))
11
12 a_doubled = numpy.empty_like(a)
13 cuda.memcpy_dtoh(a_doubled, a_gpu)
14 print a_doubled
15 print a
```

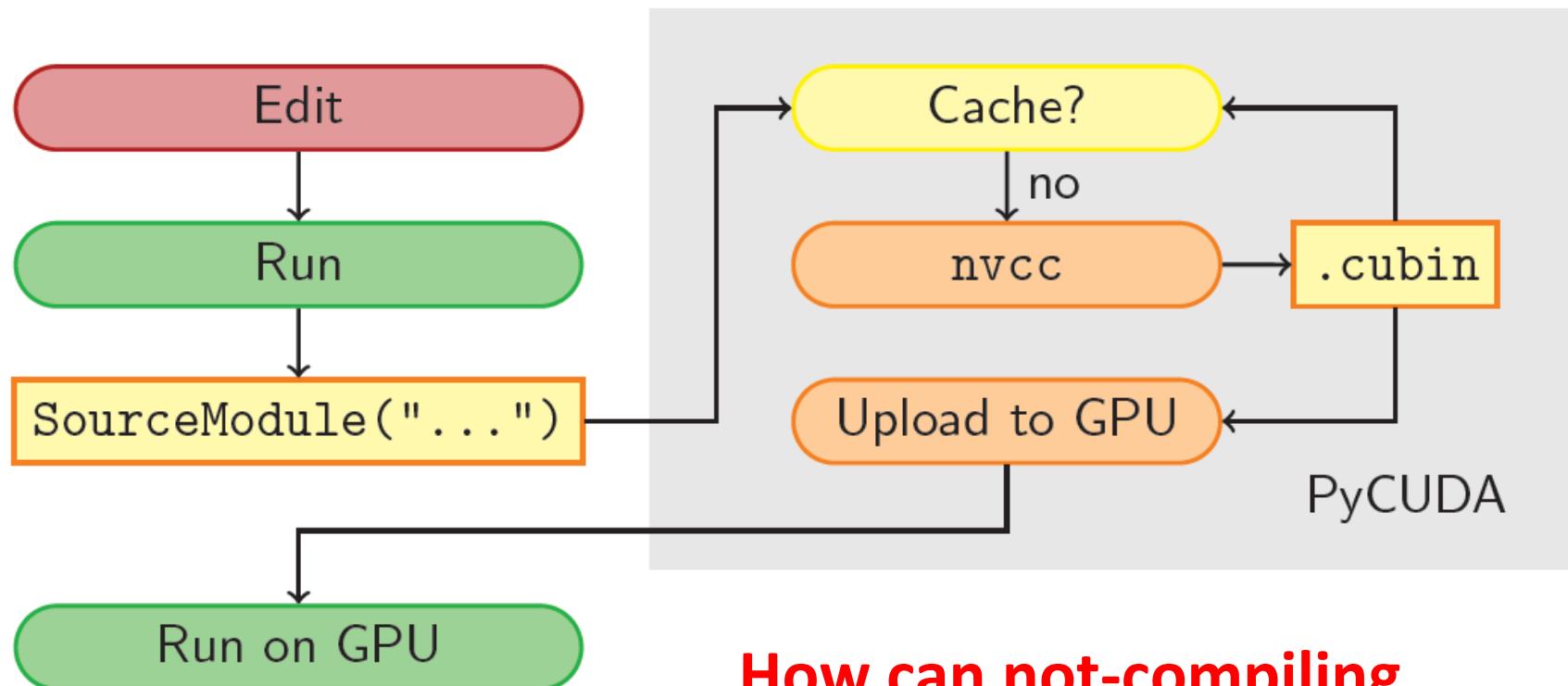


Scripting: Interpreted, not Compiled

Program creation workflow:



PyCUDA: Workflow



How can not-compiling help?

gpuarray: Reduction made easy

Example: A scalar product calculation

```
from pycuda.reduction import ReductionKernel
dot = ReductionKernel(dtype_out=numpy.float32, neutral="0",
                      reduce_expr="a+b", map_expr="x[i]*y[i]",
                      arguments="const float *x, const float *y")  
  
from pycuda.curandom import rand as curand
x = curand((1000*1000), dtype=numpy.float32)
y = curand((1000*1000), dtype=numpy.float32)  
  
x_dot_y = dot(x, y).get()
x_dot_y_cpu = numpy.dot(x.get(), y.get())
```

What does this code do?

MATLAB Parallel Computing Toolbox

```
A = gpuArray(rand(2^16,1)); % copy to GPU  
B = fft(A); % run FFT (overloaded function)  
C = gather(B); % copy back to host
```

- Only differences between this and CPU code are `gpuArray()` and `gather()`
- Can also use `arrayfun()` or your own CUDA kernel
- Any performance problems with this approach?

CUDA Libraries

- Productivity
 - Thrust
- Performance
 - cuBLAS
 - cuFFT
 - Plenty more

Prelude: C++ Templates Primer

```
template <typename T>
T sum(const T a, const T b) {
    return a + b;
}

int main() {
    cout << sum<int>(1, 2) << endl;
    cout << sum<float>(1.21, 2.43) << endl;
    return 0;
}
```

- Make functions and classes generic
- Evaluate at compile-time
- Standard Template Library (STL)
 - Algorithms, iterators, containers

Thrust - “Code at the speed of light”

- Developed by Jared Hoberock and Nathan Bell of NVIDIA Research
- Objectives
 - Programmer productivity
 - Leverage parallel primitives
 - Encourage generic programming
 - E.g. one reduction to rule them all
 - High performance
 - With minimal programmer effort
 - Interoperability
 - Integrates with CUDA C/C++ code

Thrust - Example

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>
int main(void)
{
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // sort data on the device
    thrust::sort(d_vec.begin(), d_vec.end());
    // transfer data back to host
    thrust::copy(d_vec.begin(), d_vec.end(),
                h_vec.begin());
    return 0;
}
```

Code from [GPU Computing Gems](#)

Thrust Design

- Based off STL ideas
 - Algorithms, iterators, containers
 - Generic through C++ templates
- Built on top of CUDA Runtime API
 - Ships with CUDA 4.0+
- Four fundamental parallel algorithms
 - `for_each`
 - `reduce`
 - `scan`
 - `sort`

Thrust-CUDA C Interoperability

```
size_t N = 1024;
// raw pointer to device memory
int* raw_ptr;
cudaMalloc(&raw_ptr, N*sizeof(int));
// wrap raw pointer with a device ptr
device_ptr<int> dev_ptr =
    device_pointer_cast(raw_ptr);
// use device ptr in Thrust algorithms
sort(dev_ptr, dev_ptr + N);
// access device memory through device ptr
dev_ptr[0] = 1;
// free memory
cudaFree(raw_ptr);
```

Thrust with User-Defined Functions

```
struct saxpy_functor {
    const float a;
    saxpy_functor(float a) : a(a) {}
    __host__ __device__
    float operator()(float x, float y) { return a*x+y; }
};

void saxpy(float a, device vector<float>& x, device
vector<float>& y) {
    // setup functor
    saxpy_functor func(a);
    // call transform
    transform(x.begin(), x.end(), y.begin(), y.begin(),
    func);
}
```

Thrust Performance

- Templates allow inlining and type analysis
 - How could knowing types improve global memory performance?

Table 26.1 Memory Bandwidth of Two `fill` Kernels

GPU	data type	naive fill	thrust::fill	Speedup
GeForce 8800 GTS	char	1.2 GB/s	41.2 GB/s	34.15x
	short	2.4 GB/s	41.2 GB/s	17.35x
	int	41.2 GB/s	41.2 GB/s	1.00x
	long	40.7 GB/s	40.7 GB/s	1.00x
GeForce GTX 280	char	33.9 GB/s	75.0 GB/s	2.21x
	short	51.6 GB/s	75.0 GB/s	1.45x
	int	75.0 GB/s	75.0 GB/s	1.00x
	long	69.2 GB/s	69.2 GB/s	1.00x
GeForce GTX 480	char	74.1 GB/s	156.9 GB/s	2.12x
	short	136.6 GB/s	156.9 GB/s	1.15x
	int	146.1 GB/s	156.9 GB/s	1.07x
	long	156.9 GB/s	156.9 GB/s	1.00x

Thrust Toy-box

- Kernel fusion with `transform_iterator` and `permutation_iterator`
- Conversion between arrays of structs (AoS) and structure of arrays (SoA) with `zip_iterator`
- Implicit ranges

CUDA Specialized Libraries

- NVIDIA cuBLAS
 - Basic Linear Algebra Subprograms (BLAS)
- NVIDIA cuFFT
 - Compute Fast Fourier Transforms
- NVIDIA NPP
 - Image and Signal Processing
- See more: <http://developer.nvidia.com/gpu-accelerated-libraries>

CUDA Profiling and Debugging

- Visual Profiler
- Parallel Nsight
- cuda-gdb

Visual Profiler

- Graphical profiling application
- Collects performance counter data and makes recommendations
 - Global memory throughput
 - IPC
 - Active warps/cycle
 - Cache hit rate
 - Register counts
 - Bank conflicts
 - Branch divergence
 - Many more (Full list in Visual Profiler User Guide)

Analysis for kernel fermiSgemm_v2_kernel_val on device Tesla C2050**Summary profiling information for the kernel:**

Number of calls: 31

Minimum GPU time(us): 1351.62

Maximum GPU time(us): 1362.62

Average GPU time(us): 1357.68

GPU time (%): 19.78

Grid size: [6 10 1]

Block size: [64 4 1]

Limiting Factor

Achieved Instruction Per Byte Ratio: 15.55 (Balanced Instruction Per Byte Ratio: 3.57)

Achieved Occupancy: 0.31 (Theoretical Occupancy: 0.33)

IPC: 1.73 (Maximum IPC: 2)

Achieved global memory throughput: 16.57 (Peak global memory throughput(GB/s): 144.00)

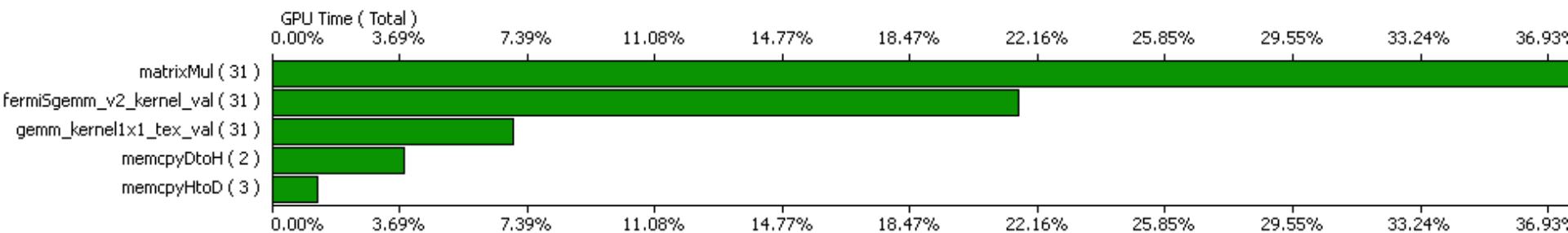
Hint(s)

- The achieved instructions per byte ratio for the kernel is greater than the balanced instruction per byte ratio for the device. Hence, the **kernel is likely compute bound**. For details, click on **Instruction Throughput Analysis**.
- The **kernel occupancy is low**. For details, click on **Occupancy Analysis**.

Limiting Factor Identification	Show all columns							
	GPU Timestamp (us)	GPU Time (us)	instructions issued Type:SM Run:8	active warps Type:SM Run:11	active cycles Type:SM Run:12	I2 read requests Type:FB	I2 read texture requests	
Memory Throughput Analysis	1 7239.42	1361.25	1082203	9282818	628510	0	920510	
Instruction Throughput Analysis	2 9046.02	1358.82	1082011	9256253	627194	0	920514	
Occupancy Analysis	3 10846	1358.59	1082112	9236393	626551	0	920538	
	4 12645.4	1358.75	1082169	9264398	627349	0	920560	
	5 14445.3	1356.58	1082026	9243036	627131	0	920520	
	6 16243.2	1359.2	1082018	11494233	625456	0	920536	
	7 18042.6	1357.86	1082063	9235858	625058	0	920564	
	8 19841.3	1359.3	1082098	9246927	626458	0	920512	
	9 21641.5	1355.87	1082060	11502813	626266	0	920546	
	10 23438.1	1356.7	1082093	9252801	626343	0	920540	
	11 25235.5	1357.47	1082120	9241326	626776	0	920540	

Profiler Output Summary Table GPU Time Summary Plot X

Gpu Time Summary Plot



Visual Profiler

Does plots too!

Parallel Nsight

- Motivation
 - Why didn't breakpoints in Visual Studio work for debugging CUDA?

Parallel Nsight

- Debugger and Profiler for:
 - CUDA
 - OpenCL
 - Direct3D Shaders
- Integrated into Visual Studio 2008/2010
- Caveat: requires extra GPU for display while debugging
 - Supports NVIDIA Optimus

voxelpipe_demo_vc10 (Debugging) - Microsoft Visual Studio (Administrator)

File Edit View Project Build Debug Team Nsight Data Tools Test Analyze Window Help

Process: [1840] voxelpipe_demo.exe Thread: [2874912] <No Name> Stack Frame: CUmodule 05508fe0 - [2] trace - Line 148

Connections: localhost

CUDA Info 1

Warp Index PC Active Mask Status Exception File Name Source Lin Lanes

Current	blockIdx	Warp Index	PC	Active Mask	Status	Exception	File Name	Source Lin	Lanes
		(0, 0, 0)	0 0x003e1ad8	0xffffffff80	Breakpoint	None	rt_render.cu	163	
		(0, 0, 0)	1 0x003e1ad8	0xffffffff00	Breakpoint	None	rt_render.cu	163	
		(0, 0, 0)	2 0x003e1ad8	0xfffffc00	Breakpoint	None	rt_render.cu	163	
		(0, 0, 0)	3 0x003e1ad8	0xfffff800	None	None	rt_render.cu	163	
		(1, 0, 0)	0 0x003e1298	0x03c00000	Breakpoint	None	rt_render.cu	148	
		(1, 0, 0)	1 0x003e1298	0x07c00000	Breakpoint	None	rt_render.cu	148	
		(1, 0, 0)	2 0x003ede70	0xffffffff	None	None	ci_include.h	423	

CUDA WarpWatch 1

Name	Type	ray_inv.x	ray_inv.y	ray_inv.z
0	_local_float	-1.4444908	-1.7955524	-2.17
1	_local_float	-1.44425	-1.7967783	-2.17
2	_local_float	-1.4440092	-1.7980076	-2.17
3	_local_float	-1.4437686	-1.7992405	-2.17
4	_local_float	-1.4435281	-1.800477	-2.17
5	_local_float	-1.4432876	-1.8017174	-2.17
6	_local_float	-1.4430474	-1.8029615	-2.17
7	_local_float	-1.4428074	-1.8042094	-2.16
8	_local_float	-1.4425675	-1.8054608	-2.16
9	_local_float	-1.4423276	-1.8067161	-2.16
10	_local_float	-1.4420878	-1.8079749	-2.16
11	_local_float	-1.4418485	-1.8092378	-2.16
12	_local_float	-1.4416089	-1.8105046	-2.16
13	_local_float	-1.4413697	-1.8117749	-2.16
14	_local_float	-1.4411306	-1.8130492	-2.16
15	_local_float	-1.4408917	-1.8143274	-2.15
16	_local_float	-1.4406527	-1.8156093	-2.15
17	_local_float	-1.4404141	-1.8168953	-2.15
18	_local_float	-1.4401754	-1.818185	-2.15
19	_local_float	-1.439937	-1.8194786	-2.15
20	_local_float	-1.4396986	-1.820776	-2.15
21	_local_float	-1.4394605	-1.8220775	-2.15
22	_local_float	-1.4392225	-1.8233831	-2.14
23	_local_float	-1.4389844	-1.8246926	-2.14
24	_local_float	-1.4387469	-1.8260059	-2.14
25	_local_float	-1.4385092	-1.8273233	-2.14
26	_local_float	-1.4382718	-1.828645	-2.14
27	_local_float	-1.4380344	-1.8299706	-2.14
28	_local_float	-1.4377974	-1.8313001	-2.14
29	_local_float	-1.4375603	-1.832634	-2.13
30	_local_float	-1.4373236	-1.8339716	-2.13
31	_local_float	-1.4370868	-1.8353136	-2.13

Disassembly

Address: 148: const uint32 leaf_index = node.get_index(

```

148: const uint32 leaf_index = node.get_index(
0x003e1298 2800400010019de4 MOV R6, c[0x0][0x4];
0x003e12a0 28000000fc01dde4 MOV R7, RZ;
0x003e12a8 28000000fc01dde4 MOV R7, RZ;
0x003e12b0 2800000018019de4 MOV R6, R6;
0x003e12b8 4801000018411c03 IADD R4,CC, R4, R6;
0x003e12c0 480000001c515c43 IADD.X R5, R5, R7;
0x003e12c8 2800000010011de4 MOV R4, R4;
0x003e12d0 2800000014015de4 MOV R5, R5;
0x003e12d8 2800000014015de4 MOV R5, R5;

```

Locals

Name	Type	Value
leaf	_local_	{m_size = 67106176, m_index = 0}
leaf_index	_local_	'leaf_index' has no value at the target location.
leaf_end	_local_	'leaf_end' has no value at the target location.
leaf_begin	_local_	'leaf_begin' has no value at the target location.
node	_local_	{m_packed_data = 214748477, m_skip_node = 24}
_T21669	_local_	{x = -1.4394605, y = -1.8220775, z = -2.150774}
ray_inv	_local_	{x = -1.4394605, y = -1.8220775, z = -2.150774}
node_index	_local_	'node_index' has no value at the target location.

Call Stack

Name	Language
CUmodule 05508fe0 - [2] trace - Line 148	CUDA
CUmodule 05508fe0 - [1] render_pixel - Line 409	CUDA
CUmodule 05508fe0 - [0] rt_trace_primary_kernel - Line 493	CUDA

Parallel Nsight, showing breakpoints for different warps, disassembled kernel code, local variables, call stack, and register values per warp

(): `info cuda threads`

**Cuda threads**

BlockIdx	ThreadIdx	To BlockIdx	ThreadIdx	Count	Virtual PC	Filename	Line
Kernel 1							
*	(0,0,0)	(0,0,0)	(0,0,0)	(0,0,0)	1 0x000000001cea9880	templates.cu	12
	(0,0,0)	(1,0,0)	(0,0,0)	(1,0,0)	1 0x000000001cea98e0	templates.cu	15
	(1,0,0)	(0,0,0)	(1,0,0)	(0,0,0)	1 0x000000001cea9880	templates.cu	12
	(1,0,0)	(1,0,0)	(1,0,0)	(1,0,0)	1 0x000000001cea98e0	templates.cu	15

```

10    T incr = this->b;
11    if (threadIdx.x == 0)
12        t += 4 + incr;
13    else
14        t += 3;
15    return t + this->a;

```

0x000000001cea9880 <_ZN8my_classIfE11my_functionEf+192>: MOV R0, R0
 0x000000001cea9888 <_ZN8my_classIfE11my_functionEf+200>: MOV R3, R3
 0x000000001cea9890 <_ZN8my_classIfE11my_functionEf+208>: MOV32I R4, 0x40800000
 0x000000001cea9898 <_ZN8my_classIfE11my_functionEf+216>: FADD R3, R3, R4
 0x000000001cea98a0 <_ZN8my_classIfE11my_functionEf+224>: FADD R0, R0, R3

Breakpoint on CUDA kernel launch at my_kernel<int, float><<(2,1,1),(2,1,1)>>> (out1=0x200100000, out2=0x200100200) at templates.cu:21
(gdb) break templates.cu:12

Breakpoint 1 at 0x1cea96f8: file templates.cu, line 12.

Breakpoint 2 at 0x1cea9880: file templates.cu, line 12.

warning: Multiple breakpoints were set.

They may be automatically deleted at the end of the run.

Use the "delete" command to delete unwanted breakpoints.

(gdb) info breakpoints

Num	Type	Disp	Enb	Address	What
1	breakpoint	keep	y	0x000000001cea96f8	in my_class<int>::my_function(int) at templates.cu:12
2	breakpoint	keep	y	0x000000001cea9880	in my_class<float>::my_function(float) at templates.cu:12

(gdb) continue

Breakpoint 1, my_class<int>::my_function (this=0x3fffc30, t=3) at templates.cu:12

(gdb) continue

Breakpoint 2, my_class<float>::my_function (this=0x3fffc38, t=2) at templates.cu:12

(gdb) where

#0 my_class<float>::my_function (this=0x3fffc38, t=2) at templates.cu:12

#1 0x000000001cea95a0 in my_kernel<int, float><<(2,1,1),(2,1,1)>>> (out1=0x200100000, out2=0x200100200) at templates.cu:29

(gdb)]

▲ Display-1: `info cuda threads` (enabled)

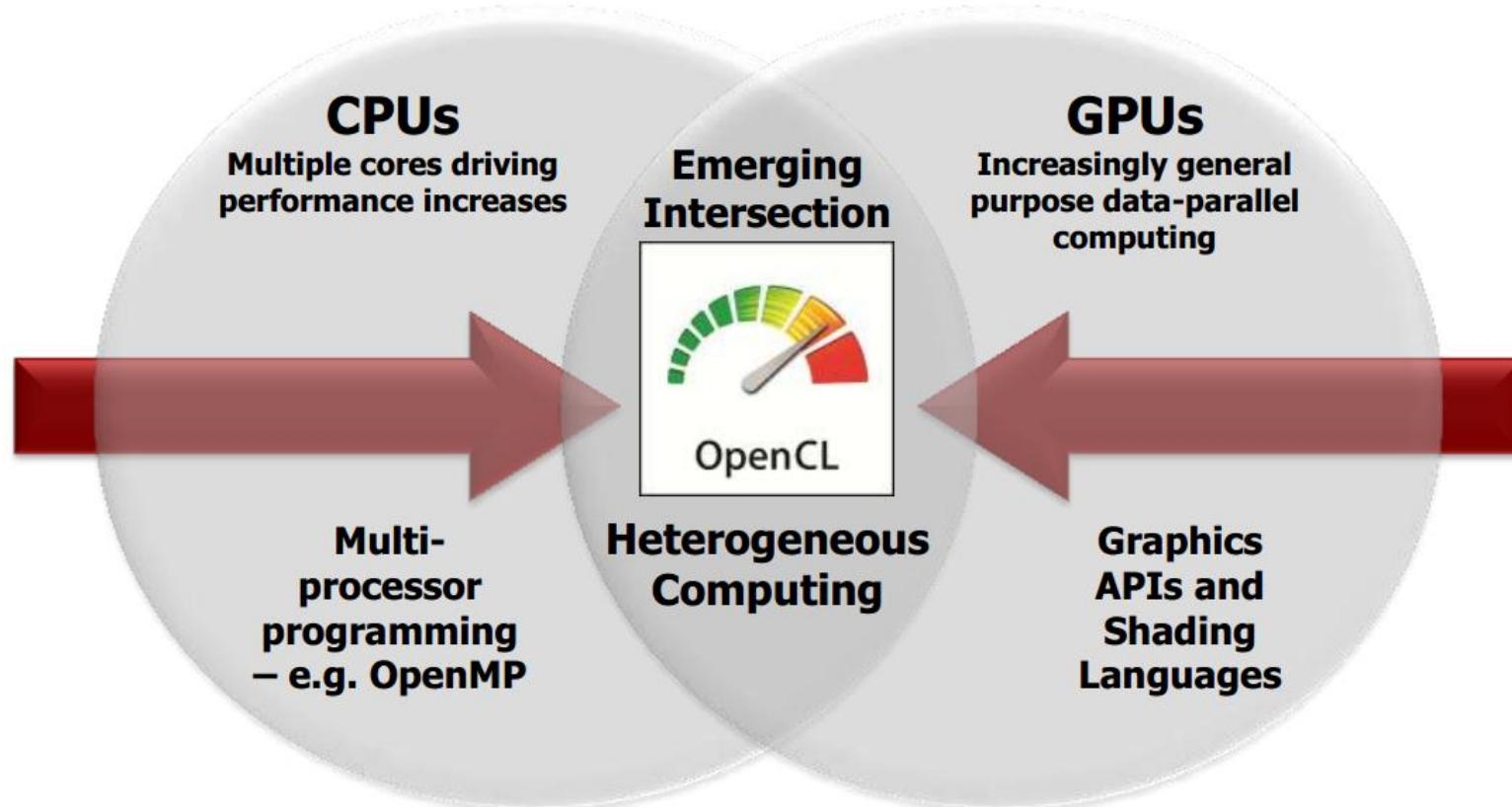


OpenCL **1.2**

OpenCL

- Initially developed by Apple with help from AMD, IBM, Intel, and NVIDIA ([Wikipedia](#))
- Specification defined by the Khronos Group

Processor Parallelism



OpenCL is a programming framework for heterogeneous compute resources

OpenCL Goals

- Parallel Compute Framework for GPUs
 - And CPUs
 - And FPGAs
 - And potentially more
- Some compliant runtimes
 - AMD APP SDK (for AMD CPUs, GPUs, and APUs)
 - Intel OpenCL SDK (for Intel CPUs)
 - NVIDIA OpenCL Runtime (for NVIDIA GPUs)

Do we want CPUs and GPUs executing the same kernels though?

OpenCL Host Code

```
size_t szLocalWorkSize[2];
size_t szGlobalWorkSize[2];

szLocalWorkSize[0] = 8;
szLocalWorkSize[1] = 8;
szGlobalWorkSize[0] = cols;
szGlobalWorkSize[1] = rows;

// setup parameter values
copyCode = oclLoadProgSource("copy_kernel.cl", "", &copyLen);
hCopyProg = clCreateProgramWithSource(t->hContext, 1, (const char **)&copyCode, &copyLen, &errcode_ret);
clBuildProgram(hCopyProg, 0, NULL, NULL, NULL, NULL);
// create kernel
t->hCopyKernel = clCreateKernel(hCopyProg, "kernel_naive_copy",
&errcode_ret);
clSetKernelArg(t->hCopyKernel, 0, sizeof(cl_mem), (void*)&dev_i_data);
clSetKernelArg(t->hCopyKernel, 1, sizeof(cl_mem), (void*)&dev_o_data);
clSetKernelArg(t->hCopyKernel, 2, sizeof(cl_int), (void*)&rows);
clSetKernelArg(t->hCopyKernel, 3, sizeof(cl_int), (void*)&cols);
clEnqueueNDRangeKernel(t->hCmdQueue, t->hCopyKernel, 2, NULL,
szGlobalWorkSize, szLocalWorkSize, 0, NULL, NULL);
```

OpenCL Host Code

```
size_t szLocalWorkSize[2];
size_t szGlobalWorkSize[2];

szLocalWorkSize[0] = 8;
szLocalWorkSize[1] = 8;      ← What are these for?
szGlobalWorkSize[0] = cols;
szGlobalWorkSize[1] = rows;

// setup parameter values
copyCode = oclLoadProgSource("copy_kernel.cl", "", &copyLen);
hCopyProg = clCreateProgramWithSource(t->hContext, 1, (const char **)&copyCode, &copyLen, &errcode_ret);
clBuildProgram(hCopyProg, 0, NULL, NULL, NULL, NULL);

// create kernel
t->hCopyKernel = clCreateKernel(hCopyProg, "kernel_naive_copy",
&errcode_ret);
clSetKernelArg(t->hCopyKernel, 0, sizeof(cl_mem), (void*)&dev_i_data);
clSetKernelArg(t->hCopyKernel, 1, sizeof(cl_mem), (void*)&dev_o_data);
clSetKernelArg(t->hCopyKernel, 2, sizeof(cl_int), (void*)&rows);
clSetKernelArg(t->hCopyKernel, 3, sizeof(cl_int), (void*)&cols);
clEnqueueNDRangeKernel(t->hCmdQueue, t->hCopyKernel, 2, NULL,
szGlobalWorkSize, szLocalWorkSize, 0, NULL, NULL);
```

Look Familiar?



OpenCL Device Code

```
__kernel void kernel_naive_copy(
    __global const float4 * i_data,
    __global float4 * o_data,
    int rows, int cols)
{
    uint x = get_global_id(0);
    uint y = get_global_id(1);
    o_data[y*rows + x] = i_data[y*rows + x];
}
```

See some similarities?

OpenCL Code

- Very similar to CUDA Driver API and CUDA C
 - NVIDIA has a short [guide](#) outlining syntax differences
- C-based API
 - C++ wrappers and bindings to other languages (e.g. PyOpenCL) available

Which should I choose?

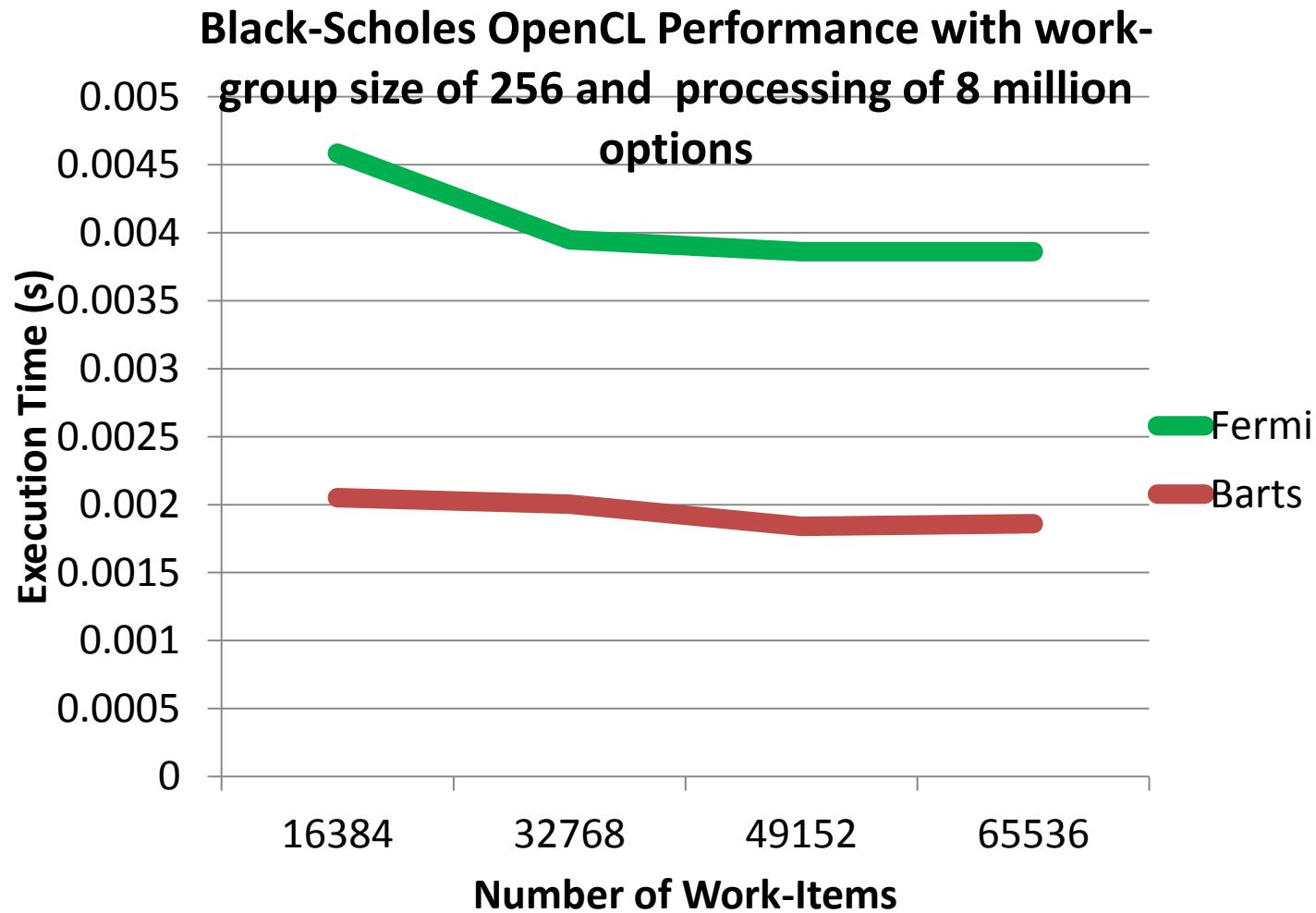
OPENCL OR CUDA?

Compatibility

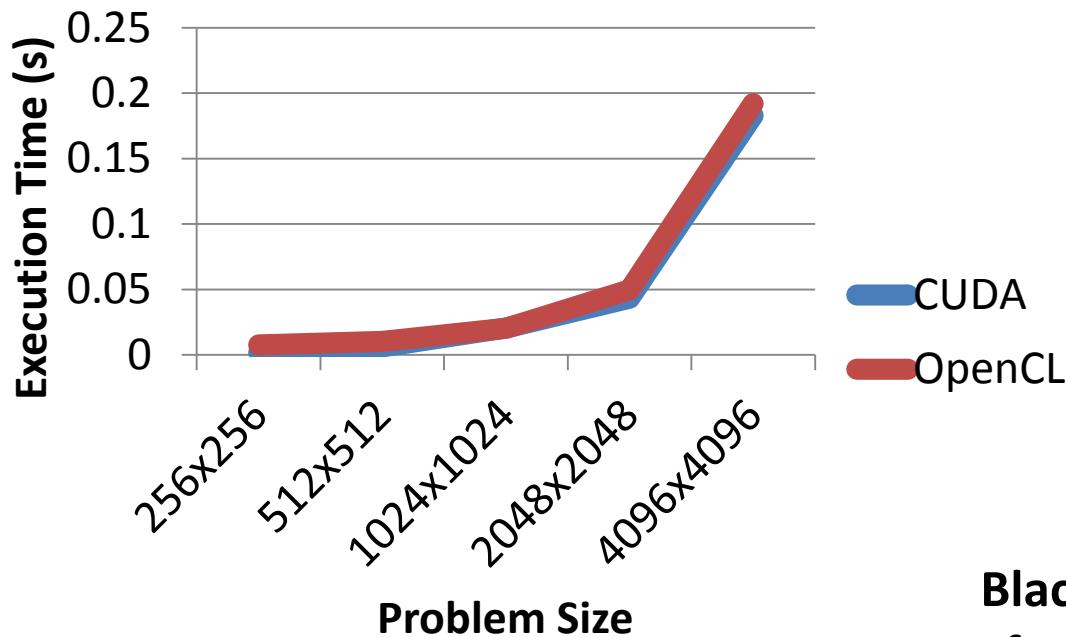
- CUDA runs only on NVIDIA GPUs
 - Not necessarily true...
- OpenCL is supported by a lot of vendors



Doesn't everyone just want an NVIDIA GPU?



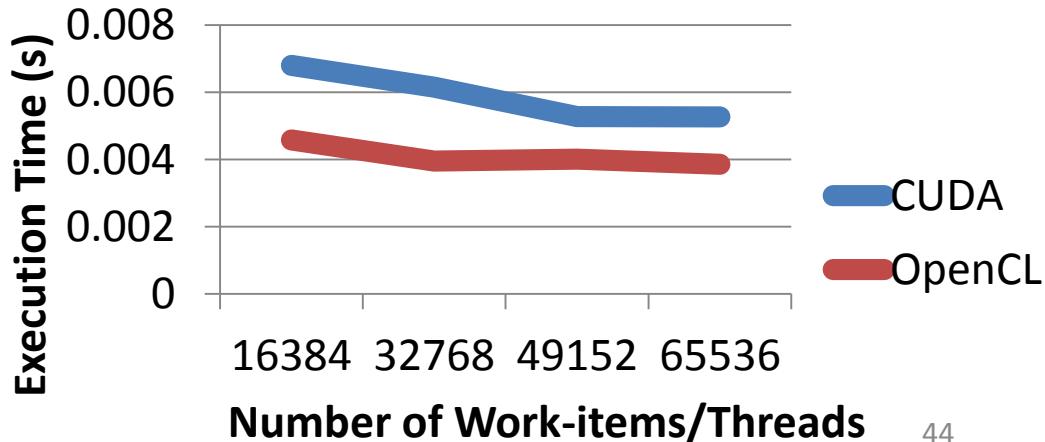
SAT OpenCL and CUDA Performance with work-group size of 256



Performance Comparison on NVIDIA GPUs

[This was done with the CUDA 3.2 Toolkit. CUDA 4.1 brought a new LLVM compiler to CUDA (OpenCL compiler was already LLVM-based)]

Black-Scholes OpenCL and CUDA Performance with work-group size of 256 and processing of 8 million options



Programming Framework Comparison

- CUDA 4.0 brought a lot of advancements
 - Unified address space
 - C++ new/delete, virtual functions on device
 - GPUDirect peer-to-peer GPU communication
- OpenCL does not have these features
 - And 18-month release cadence is slower than NVIDIA's

Libraries & Mindshare

- CUDA has a larger ecosystem
 - Thrust is a particularly important library
- Will OpenCL catch up?
 - Growing in other ways
 - OpenCL Embedded Profiles
 - WebCL

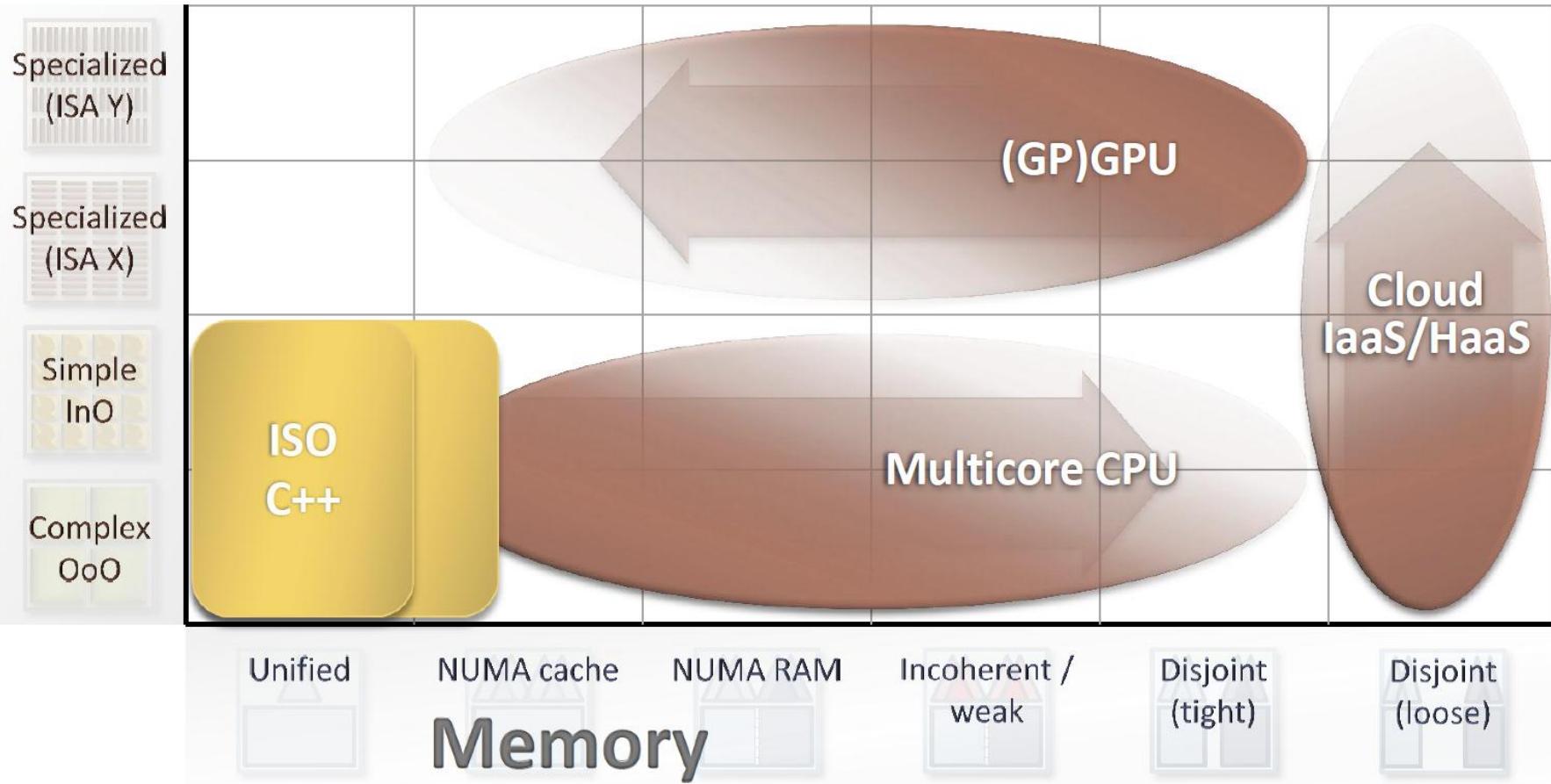
C++ AMP

C++ AMP (Accelerated Massive Parallelism)

- Announced by Microsoft in June 2011
- Targeting “heterogeneous parallel computing”
 - Multicore CPUs
 - GPUs
 - Cloud Infrastructure-as-a-Service (IaaS)

Programming Models & Languages

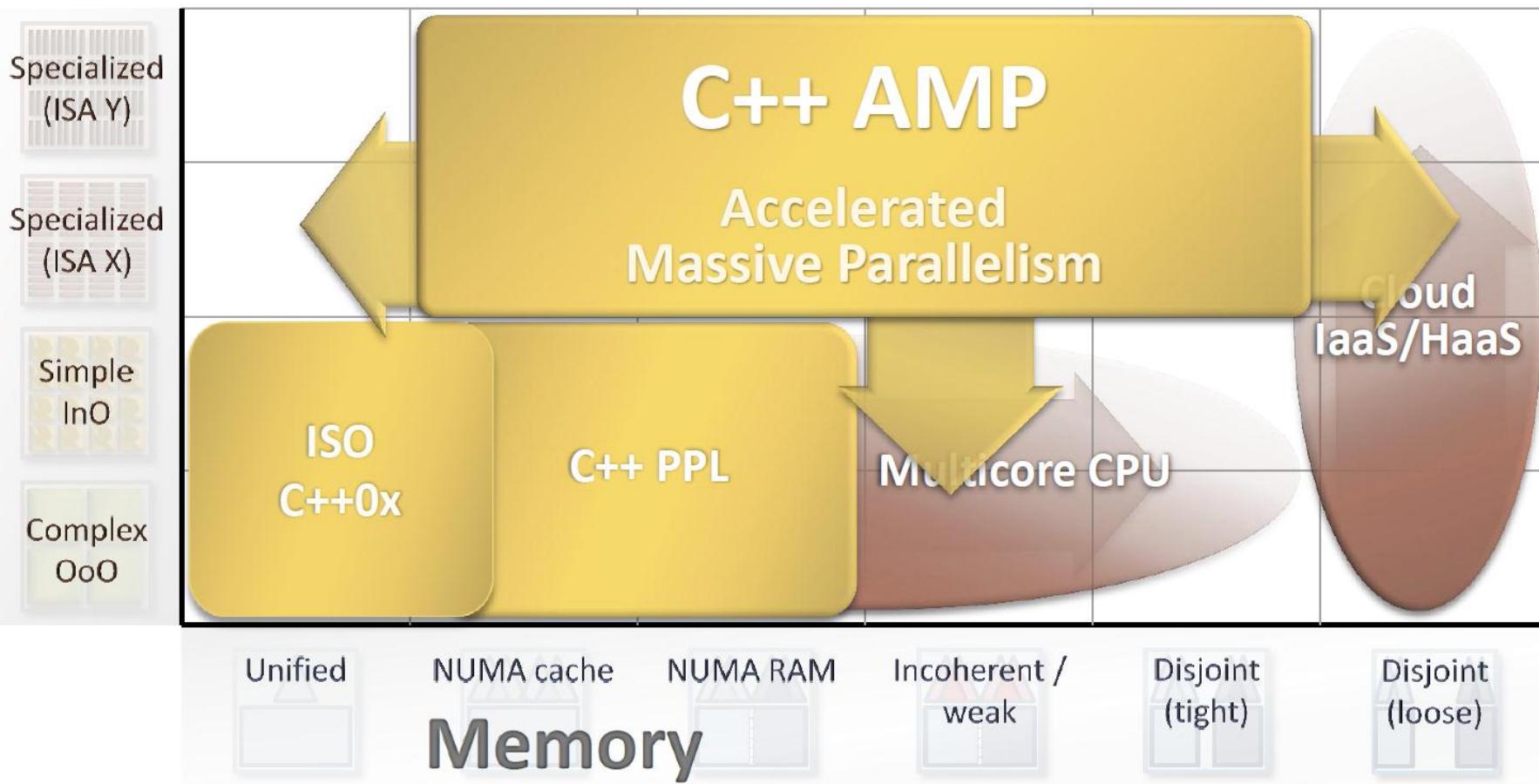
Processors



Slide from Herb Sutter's AMD
Fusion [Keynote](#)

Programming Models & Languages

Processors



Slide from Herb Sutter's AMD
Fusion [Keynote](#)

C++ AMP Matrix Multiply

```
void MatrixMult( float* C, const vector<float>& A,
const vector<float>& B,
int M, int N, int W )
{
    array_view<const float,2> a(M,W,A), b(W,N,B);
    array_view<writeonly<float>,2> c(M,N,C);
    parallel for each( c.grid, [=] (index<2> idx)
        restrict(direct3d) {
            float sum = 0;
            for(int i = 0; i < a.x; i++)
                sum += a(idx.y, i) * b(i, idx.x);
            c[idx] = sum;
        } );
}
```

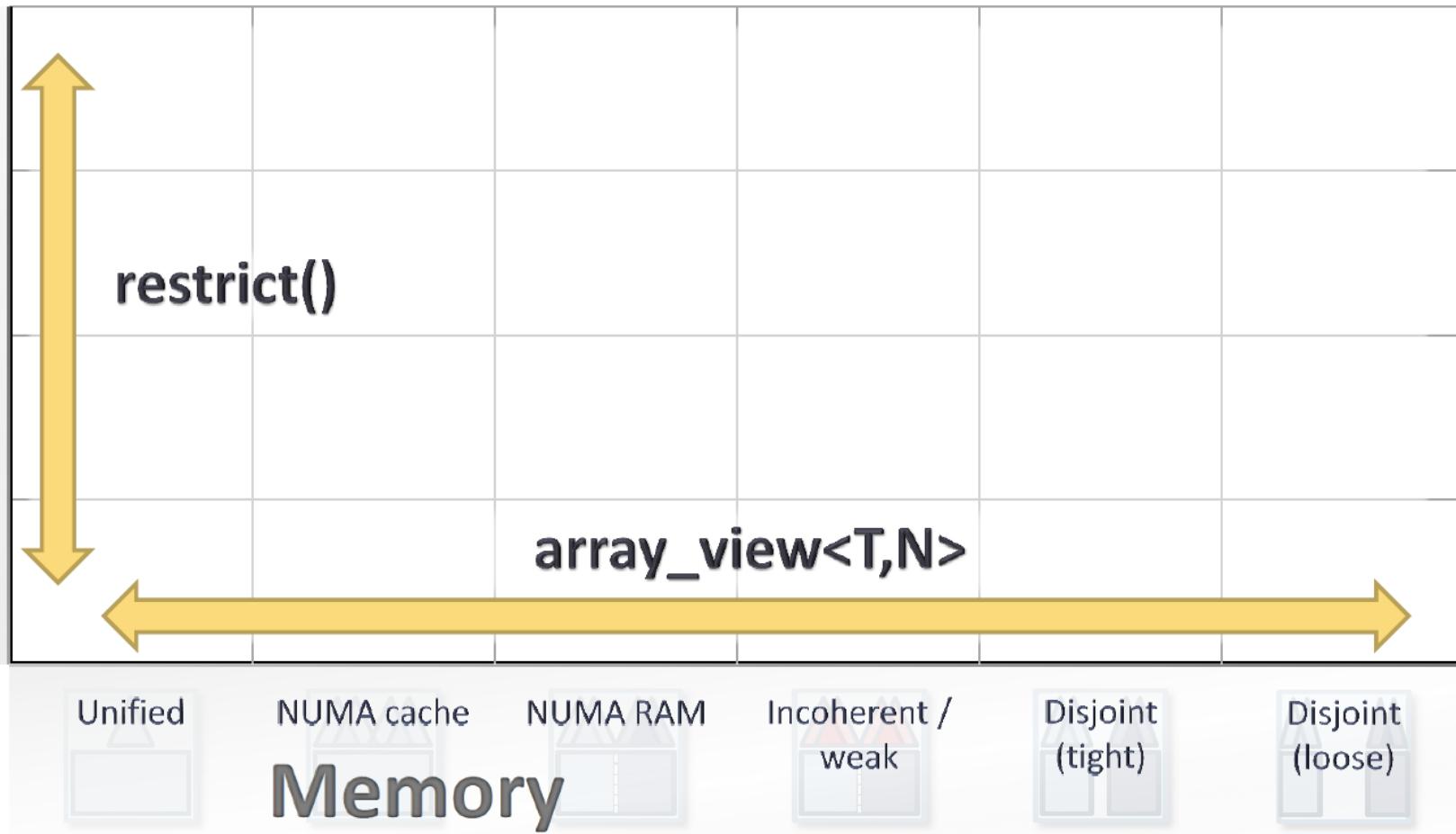
C++ AMP Matrix Multiply

```
void MatrixMult( float* C, const vector<float>& A,
const vector<float>& B,
int M, int N, int W )
{
    array_view<const float,2> a(M,W,A), b(W,N,B);
    array_view<writeonly<float>,2> c(M,N,C);
    parallel for each( c.qgrid, [=] (index<2> idx)
        restrict(direct3d) {
            float sum = 0;
            for(int i = 0; i < a.x; i++)
                sum += a(idx.y, i) * b(i, idx.x);
            c[idx] = sum;
        } );
}
```

- `array_view`: abstraction for accessing data (like an “iterator range”)
- `Lambda expressions`: like functors of thrust but with less syntactic overhead
- `restrict`: ensure only language capabilities supported by device are used

C++ AMP at a Glance

Processors

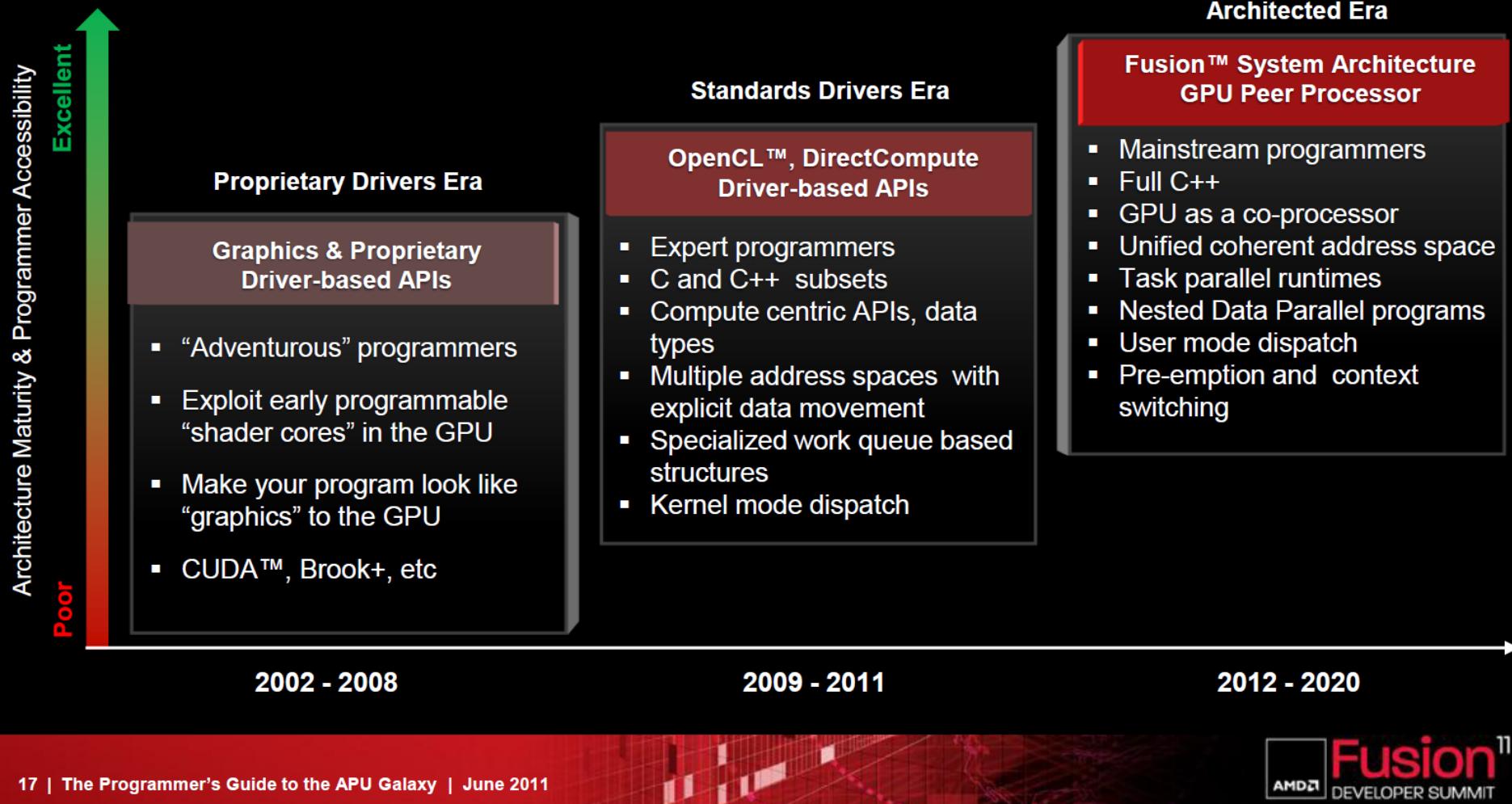


Slide from Herb Sutter's AMD
Fusion [Keynote](#)

C++ AMP

- Only 1 new keyword added to C++
 - All other functionality in classes and functions
- Released as open specification 2 weeks ago
- Debugging and Profiling included in Visual Studio 11

EVOLUTION OF HETEROGENEOUS COMPUTING



Conclusion: What do you think?

References

- Bell, Nathan and Hoberock, Jared. “Thrust: A Productivity-Oriented Library for CUDA.” GPU Computing Gems: Jade Edition. [Link](#)
- Klöckner, Andreas. “PyCUDA: Even Simpler
- GPU Programming with Python.” [Slides](#)
- Reese, Jill and Zaranek, Sarah. “GPU Programming in MATLAB.” [Link](#)
- Rosenberg, Ofer. “OpenCL Overview.” [Slides](#)
- Sutter, Herb. “Heterogeneous Parallelism at Microsoft.” [Link](#)

Bibliography

- Klöckner, et al. “PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation.” [arXiv](#)
- Moth, Daniel. “Blazing-fast code using GPUs and more, with C++ AMP.” [Link](#)