



NVIDIA Update

June 2010



Outline

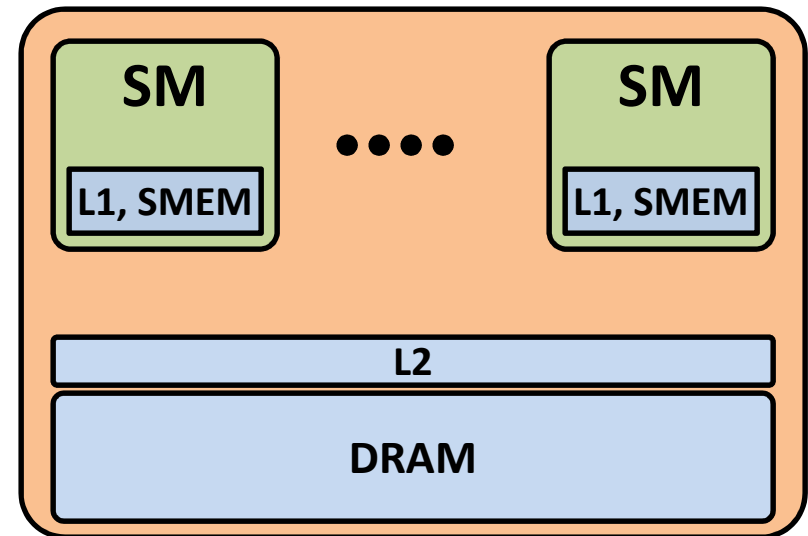
- **Hardware:**
 - GPU Hardware in General
 - Fermi Review
- **Software:**
 - Programming Languages
 - Programming Tools
 - Libraries



HARDWARE

GPU from a High Level

- **GPU:**
 - A number of multiprocessors (several to tens)
 - Memory
- **Each multiprocessor (SM) has its own:**
 - Control unit:
 - Decodes/issues instructions
 - Schedules threads
 - Pipelines for execution:
 - int, fp32, fp64, special-function, etc.
 - Registers
 - Shared memory (SMEM)
 - Programmer-managed scratch-pad memory
 - L1 data cache, constant cache
 - Hw-managed caches
 - Texture unit
 - Hw-managed cache for textures
 - Interpolation/conversion unit



Not to scale
Previous to Fermi: no L1, L2



GPU Instruction Issue

- **In-order issue**
- **Instructions are issued per *warp***
 - Warp = **32** consecutive threads
 - Think: **32-thread** vector
 - Maximum perf: **32** threads execute the same instruction
 - Each thread **CAN** execute its own code path
 - HW handles branching/predication
 - Performance reduction when paths diverge within a warp
 - Entire warp must “execute” the paths taken by threads in a warp
 - No performance reduction if:
 - All threads in a warp branch together
 - Threads in different warps take different code paths



Registers and Warp Scheduling

- **Registers (and other thread state) are partitioned among threads**
 - No overhead to store/restore state when switching threads
- **Instruction and memory latency is hidden by:**
 - Switching threads
 - Not caches, like on CPUs
 - **100s** of concurrent threads per multiprocessor
 - Run-time scheduled at warp granularity
 - Think: extreme hyperthreading
 - Independent instructions from the same thread
 - Thread blocks when instruction argument isn't available

Memory Access

- **Memory accesses are issued per warp**
 - Think: vector of 32 addresses
- **A bus connects memory and multiprocessors**
 - Bus transactions are contiguous, aligned memory regions
 - Transaction size varies from 32 to 128 bytes
- **Throughput**
 - Maximum throughput when a warp addresses and utilizes entire regions
 - Arbitrary addressing patterns within warp **ARE** allowed
 - Performance degradation due to multiple partially-used transactions per access



Fermi Multiprocessor (1)

- **Up to 1536 concurrent threads**
- **32K 32-bit registers**
 - Up to **63** registers per thread (21 if threads fully populated)
- **Instruction throughput:**
 - 2 fp32 pipes, each: 1 warp (32 threads) / 2 cycles
 - 2 int32 pipes, each : 1 warp / 2 cycles
 - fp64: 1 warp / 4 cycles
 - SFU: 1 warp / 16 cycles
- **Dual-issue:**
 - Greatly improved over previous generation GPUs
 - Instructions from 2 different warps to 2 different pipes



Fermi Multiprocessor (2)

- **64K total smem/L1**
- **Programmer chooses smem/L1 breakdown:**
 - 48KB smem, 16KB L1
 - 16KB smem, 48KB L1
- **Shared memory**
 - 32 banks, each 32 bits wide (Previous generation had 16 banks)
 - Improvements over previous generation:
 - Multicasting (previously only broadcast)
 - No bank-conflicts when accessing 64-, 128-bit words
- **L1 cache**
 - 128-byte line
 - Programmer can choose a path (compiler flag) where access granularity is 32B (but without the possibility of hitting in L1)
 - Not coherent across multiprocessors



Fermi Multiprocessor (3)

- **Addressing:**
 - Generic address space
 - Same assembly instruction can be used for gmem, smem, ...
 - Prev gen needed specific instructions for each memory
 - Enables function calling on GPU
 - 32-bit or 64-bit addresses
 - Default based on the OS
- **64-bit integer arithmetic**
 - No native int64 instructions
 - But there is magic to minimize needed int32 instructions

Fermi Multiprocessor (4)

- **64KB of constant memory, as in prev gen**
 - Declare with `__constant__` qualifier
 - Cached, read-only access (4B / warp / SM)
- **Addition: uniform memory access**
 - Uniform = same across a threadblock
 - For example: stencil coefficients for FD
 - Can be used for any address in global memory
 - Thus: no limit on size, no need for `__constant__` declaration
 - Cached (read-only, write-unaware cache, 4B / warp / SM)
 - Requirements:
 - Kernel pointer argument must be `const` qualified
 - Dereferencing must not depend on thread ID

Fermi Memory

- **L1 cache described under Multiprocessor (2)**
- **L2 cache:**
 - 768 KB per GPU
 - Coherent across multiprocessors and CPU
 - All accesses to GPU DRAM go through L2
 - Including CPU-GPU memcopies
- **GPU memory (DRAM)**
 - 6 partitions of GDDR5
 - Additional improvements over previous generation:
 - available ECC (driver option)



Fermi C2050

- **14 multiprocessors**
- **Clocks:**
 - Core: **1150 MHz**
 - Mem: **1500 MHz**
- **Throughputs:**
 - Instruction: **515 Ginstr/s**
 - Fp32: **1030 Gflops/s** peak
 - Fp64: **515 Gflops/s** peak
 - Memory:
 - Shared memory: **1030 GB/s** aggregate
 - L1: **1030 GB/s** aggregate
 - L2: **230 GB/s** (not affected by ECC)
 - DRAM: **144 GB/s** (**115 GB/s** with ECC)
- **Perfect balance:**
 - **3.5 : 1** instructions : gmem bytes (**4.5 : 1** if ECC is on)
 - Higher = instruction bound; lower = memory bound



Additional New Capabilities

- **Fermi can execute several kernels concurrently**
 - Threadblocks from one kernel are launched first
 - If there are resources available, threadblocks from a kernel in another stream are launched
- **Fermi has 2 copy-engines**
 - Can concurrently copy CPU-GPU and GPU-CPU across PCIe
 - PCIe is duplex, so aggregate bandwidth is doubled in such cases
 - Previous generation could only do one copy
- **ECC:**
 - DRAM, L2, L1, shared memory

Programming for Fermi

- Largely the same as for previous generations
- Biggest difference: memory is accessed in groups of **32** threads (previously 16), matches instruction-issue width
 - The **32** addresses of a warp would ideally fully utilize cache line(s): address a contiguous, aligned region
 - What this means to code:
 - 2D/3D threadblocks should be a multiple of **32** “wide”
 - Data should be a multiple of **32** in the fastest-varying dimension
- **L1/L2 Caches:**
 - Not designed for CPU-style reuse, so don’t worry about blocking
 - If you can block, you should be blocking for shared memory
 - Designed to improve perf for misaligned access, small strides, some register spilling
 - Some knobs to experiment with:
 - 16KB vs 48 KB L1 (configurable from source code per kernel)
 - 32B vs 128B “granularity” (only 128B allows hitting in L1), compiler option



SOFTWARE



Software

- **Languages and APIs**
 - CUDA C (C++)
 - CUDA Fortran
 - OpenCL
 - DirectCompute
- **Developer tools**
 - Debuggers
 - Profilers
 - IDEs
 - Libraries

CUDA

GPU Computing Applications

CUDA C/C++

- Over 60,000 developers
- Running in Production since 2008
- SDK + Libs + Visual Profiler and Debugger

OpenCL

- 1st GPU demo
- Shipped 1st OpenCL Conformant Driver
- Public Availability
- SDK + Visual Profiler

Direct Compute

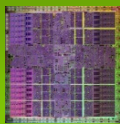
- Microsoft API for GPU Computing
- Supports all CUDA-Architecture GPUs (DX10 and DX11)

Fortran

- PGI Accelerator
- PGI CUDA Fortran

Python, Java, .NET, ...

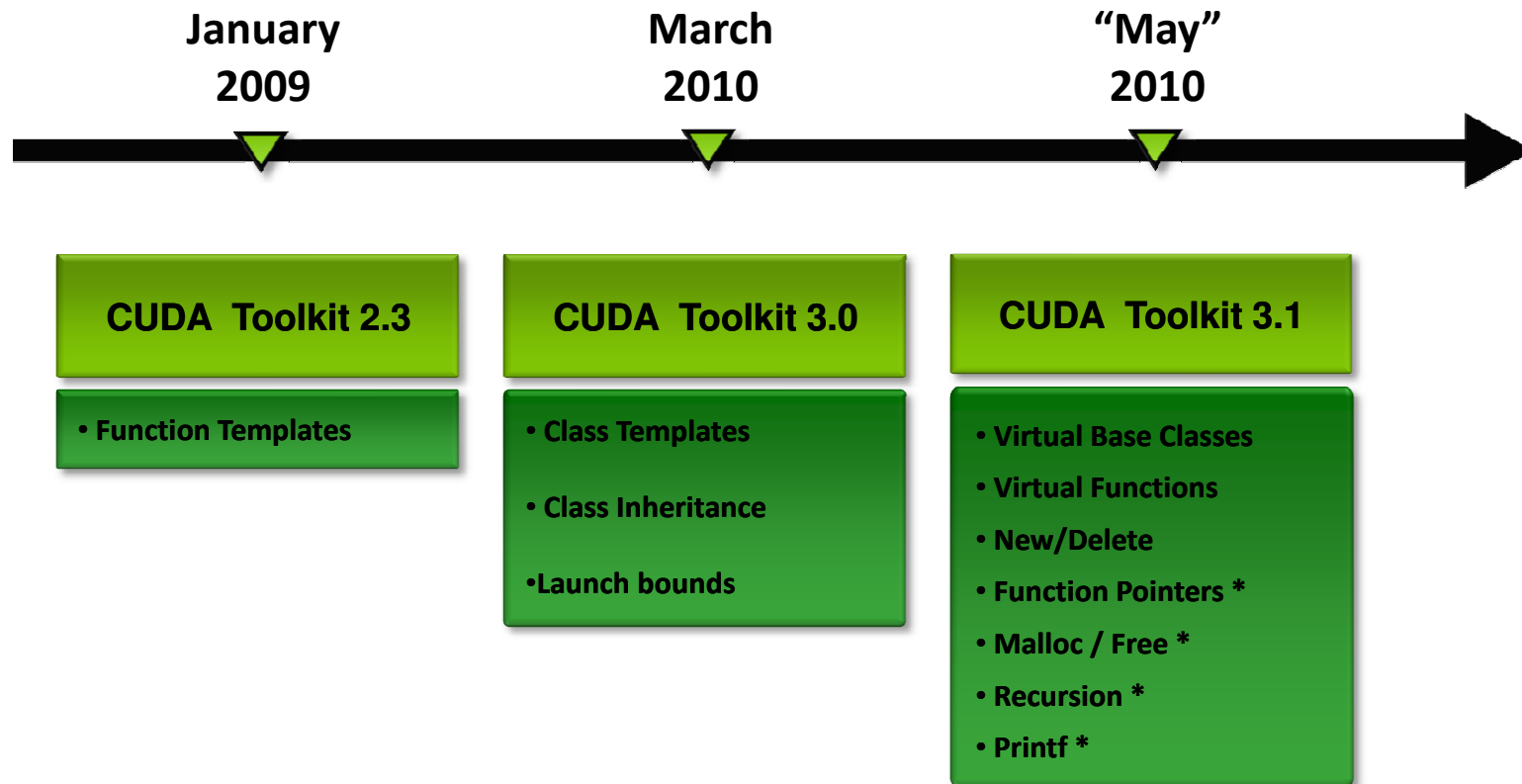
- PyCUDA
- jCUDA
- CUDA.NET
- OpenCL.NET



NVIDIA GPU

with the CUDA Parallel Computing Architecture

CUDA C Compiler Features



* Enhancement to CUDA C



CUDA C++

- **C++ Features:**
 - Classes with static methods (mostly) worked prior to CUDA 3.0
 - CUDA 3.0 officially supports inheritance and templates
 - More C++ functionality is coming in CUDA 3.1 and later
 - Note that calls like new/delete will be functional, but performance could suffer:
 - Memory is an atomic resource, you likely have 1000s of concurrent threads
- **One caveat:**
 - Methods must be qualified with `__device__` for execution on GPU
 - There is no perfect solution:
 - Not all methods written for CPU can be compiled for GPU
 - Think: methods that spawn pthreads, open windows, open sockets, etc.
 - Compiling only the needed methods can cause problems with dynamic linking

CUDA Fortran

- **CUDA Fortran**

- Collaboration between PGI and NVIDIA
- Host and GPU (kernel) code is in Fortran
 - Host (CPU) code similar to CUDA C run-time
 - Some CUDA C features are still missing
- Strongly typed (the location of data)
 - No need for special allocation/deallocation
 - Transfers between CPU and GPU are initiated by assignments

- **PGI Accelerator**

- Source directives (a la OpenMP), auto-generated kernels
- CAPS HMPP is another such tool

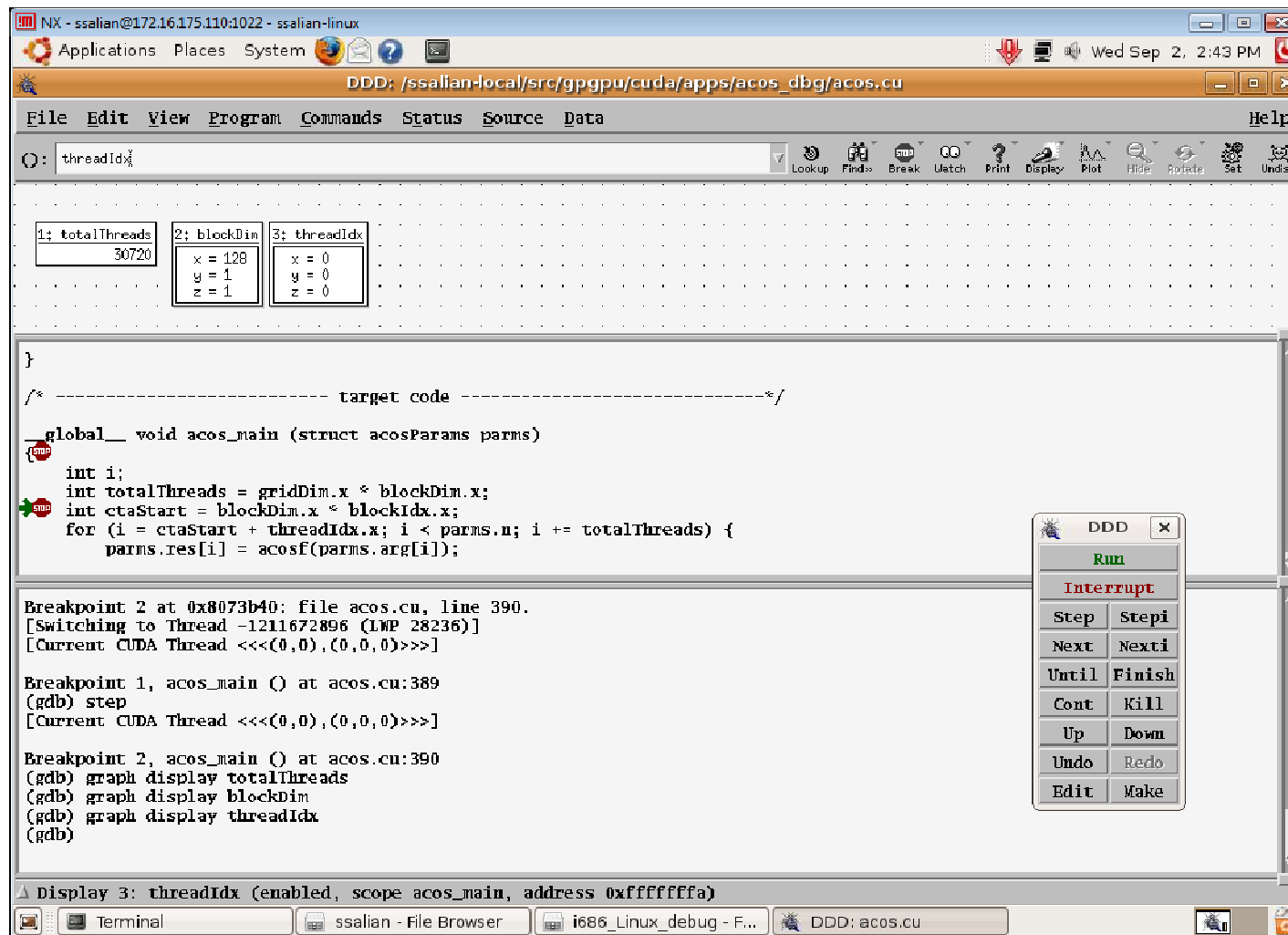
<http://www.pgroup.com/resources/cudafortran.htm>



CUDA-GDB

- **Executes on GPU hardware**
 - Not in emulation on CPU
- **Support**
 - Currently in beta, supports CUDA C
 - Integrated into GDB on all supported Linux distributions
- **What you would expect from a debugger:**
 - Breakpoints
 - Step-through the code
 - Inspect the contents of variables and memory
- **Can be used from:**
 - DDD, Emacs, ...
- **Upcoming features:**
 - Machine assembly debugging
 - Performance improvements

CUDA-GDB



NX - ssalian@172.16.175.110:1022 - ssalian-linux
 Applications Places System Wed Sep 2, 2:43 PM
 DDD: /ssalian-local/src/gpgpu/cuda/apps/acos_dbg/acos.cu

File Edit View Program Commands Status Source Data Help
 Q: threadIdx

1: totalThreads 30720	2: blockDim x = 128 y = 1 z = 1	3: threadIdx x = 0 y = 0 z = 0
--------------------------	--	---

```

}
/* ----- target code ----- */
global__ void acos_main (struct acosParams parms)
{
  int i;
  int totalThreads = blockDim.x * blockDim.x;
  int ctaStart = blockDim.x * blockIdx.x;
  for (i = ctaStart + threadIdx.x; i < parms.n; i += totalThreads) {
    parms.res[i] = acosf(parms.arg[i]);
  }
}
  
```

Breakpoint 2 at 0x8073b40: file acos.cu, line 390.
 [Switching to Thread -1211672896 (LWP 28236)]
 [Current CUDA Thread <<<(0,0),(0,0,0)>>>]

Breakpoint 1, acos_main () at acos.cu:389
 (gdb) step
 [Current CUDA Thread <<<(0,0),(0,0,0)>>>]

Breakpoint 2, acos_main () at acos.cu:390
 (gdb) graph display totalThreads
 (gdb) graph display blockDim
 (gdb) graph display threadIdx
 (gdb)

Display 3: threadIdx (enabled, scope acos_main, address 0xffffffff)

Terminal | ssalian - File Browser | i686_Linux_debug - F... | DDD: acos.cu

DDD x
 Run
 Interrupt
 Step StepI
 Next NextI
 Until Finish
 Cont Kill
 Up Down
 Undo Redo
 Edit Make

CUDA-MemCheck



- **Detects/tracks memory errors**
 - Out of bounds accesses
 - Misaligned accesses (types must be aligned on their size)
- **Integrated into CUDA-GDB**
- **Linux and WinXP**
- **Win7 and Vista support coming**

```
[jchase@dhcp-██████████i686_Linux_debug]$ cuda-memcheck ./ptrchecktest
===== CUDA-MEMCHECK
Checking...
Done
Checking...
Error: 3 (65538)
Done
Checking...
Error: 0 (1)
Error: 1 (0)
Error: 2 (0)
Done
unspecified launch failure : 125
===== Invalid read of size 4
=====   at 0x000000f0 in kernel2 (/src/gpgpu/cudamemcheck/test/ptrchecktest.cu:27)
=====   by thread 5 in block 3
===== Address 0x00101015 is misaligned
=====
===== Invalid read of size 4
=====   at 0x000000f0 in kernell (/src/gpgpu/cudamemcheck/test/ptrchecktest.cu:18)
=====   by thread 3 in block 5
===== Address 0x00101028 is out of bounds
=====
```




CUDA/OpenCL Profiling

- **Hardware counters**
 - Memory transactions, instructions, hits/misses, ...
 - Counted per multiprocessor(s), extrapolated to full GPU
- **Timing:**
 - Kernel/memcopy times
 - Kernel/memcopy timestamps
 - Kernel occupancy
- **Command-line profiler**
 - Part of the GPU driver (all OSs)
 - Enabled/disabled via environment variables
 - Configured via a file, output to a logfile
 - 4 counters per run (GT200)



CUDA/OpenCL Profiling

- **Visual Profiler**

- GUI included with the toolkit download (all OSs)
- Automatically runs the app multiple times when more than 4 counters are selected
- Derives throughput measurements from counters
 - Memory throughput, instruction throughput
- Various graphical plots of kernels, memcopies

	Method	GPU usec	%GPU time	glob mem read throughput (GB/s)	glob mem write throughput (GB/s)	glob mem overall throughput (GB/s)	instruction throughput
1	fwd_3D_16x16_order8	3.09382e+06	82.15	46.9465	11.6771	58.6236	0.763973
2	memcpyHtoD	503094	13.35				
3	memcpyDtoH	168906	4.48				



Nsight

- **A full Visual-Studio integrated development environment**
 - CUDA project build system
 - Parallel debugger
 - System analyzer
 - Graphics inspector
- **Supports:**
 - CUDA C, OpenCL, DirectCompute
 - DirectX 10, DirectX 11, OpenGL
- **Requires:**
 - Windows 7 or Vista
 - Visual Studio 2008 SP1
- **Analyze/debug:**
 - Same workstation (requires 2 GPUs)
 - Remotely (across network)





Nsight GPU Debugging

- **Executes on GPU**
 - Not emulation on CPU
- **Currently supports code written in:**
 - CUDA C
 - HLSL
- **Provides what you'd expect from a debugger:**
 - Breakpoints
 - Variable and memory inspection
- **Enhancements for parallel code:**
 - Conditional breakpoints
 - Use thread/block ID, variable values
 - Choose to break when condition is true or when it changes
 - Inspection navigation across threads/blocks

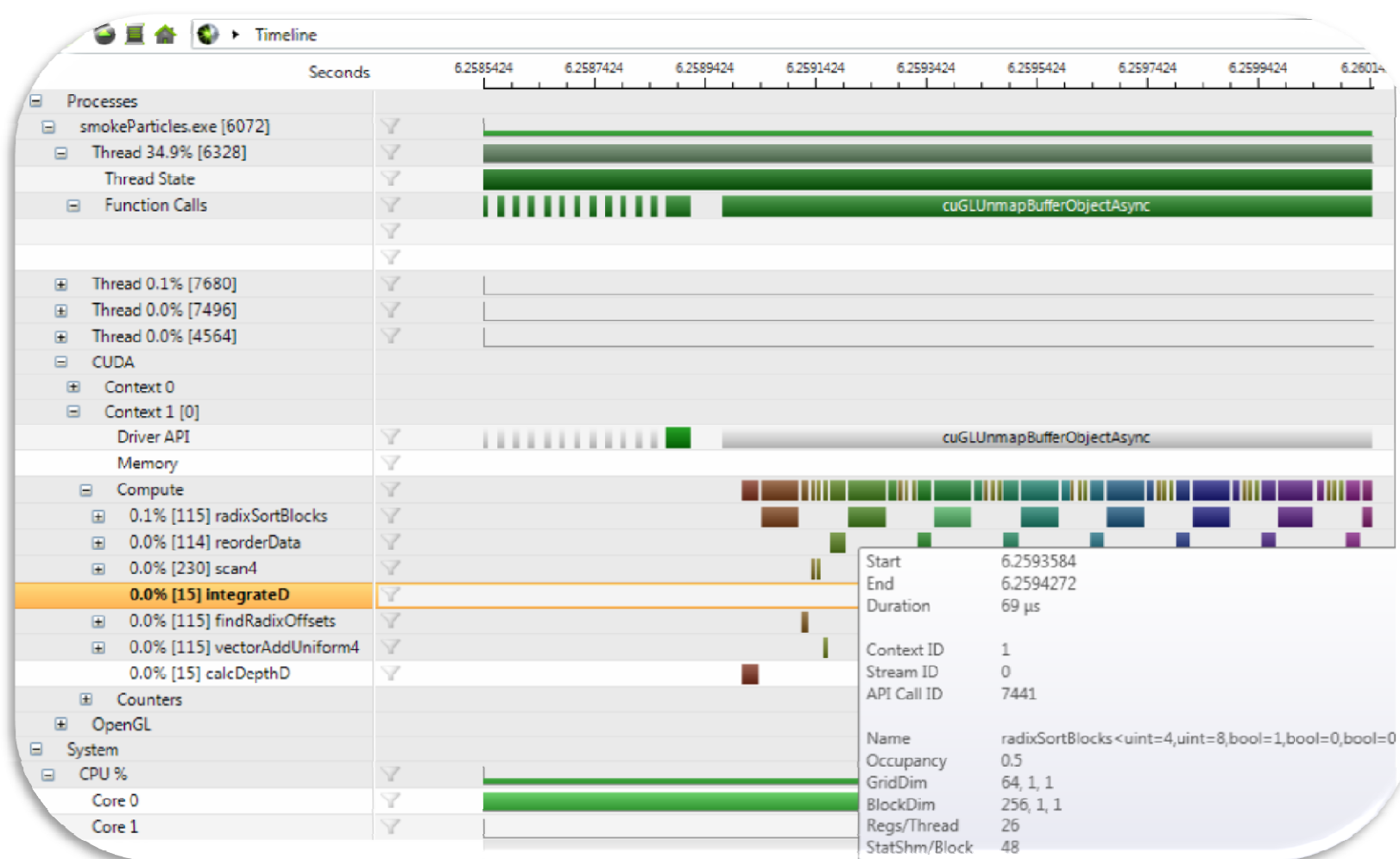


Nsight Analysis

- **Capture data for:**
 - CPU threads
 - CUDA/OpenCL calls
 - Graphics API calls
- **View a correlated trace timeline of CPU and GPU events**
- **Get profiler information for every kernel**
 - GPU counters and derived analysis

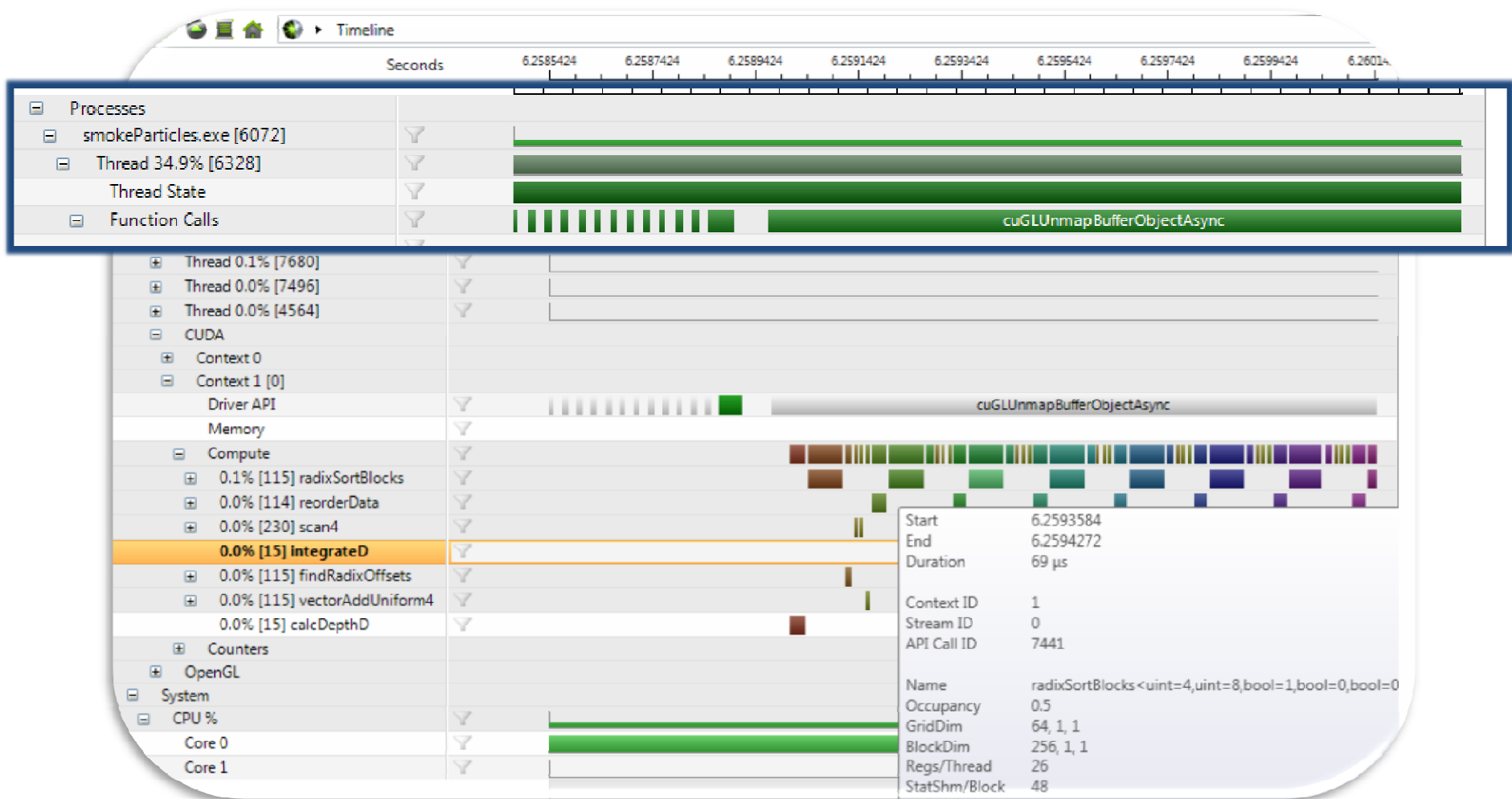


Nsight Analysis: Correlated Trace Plot



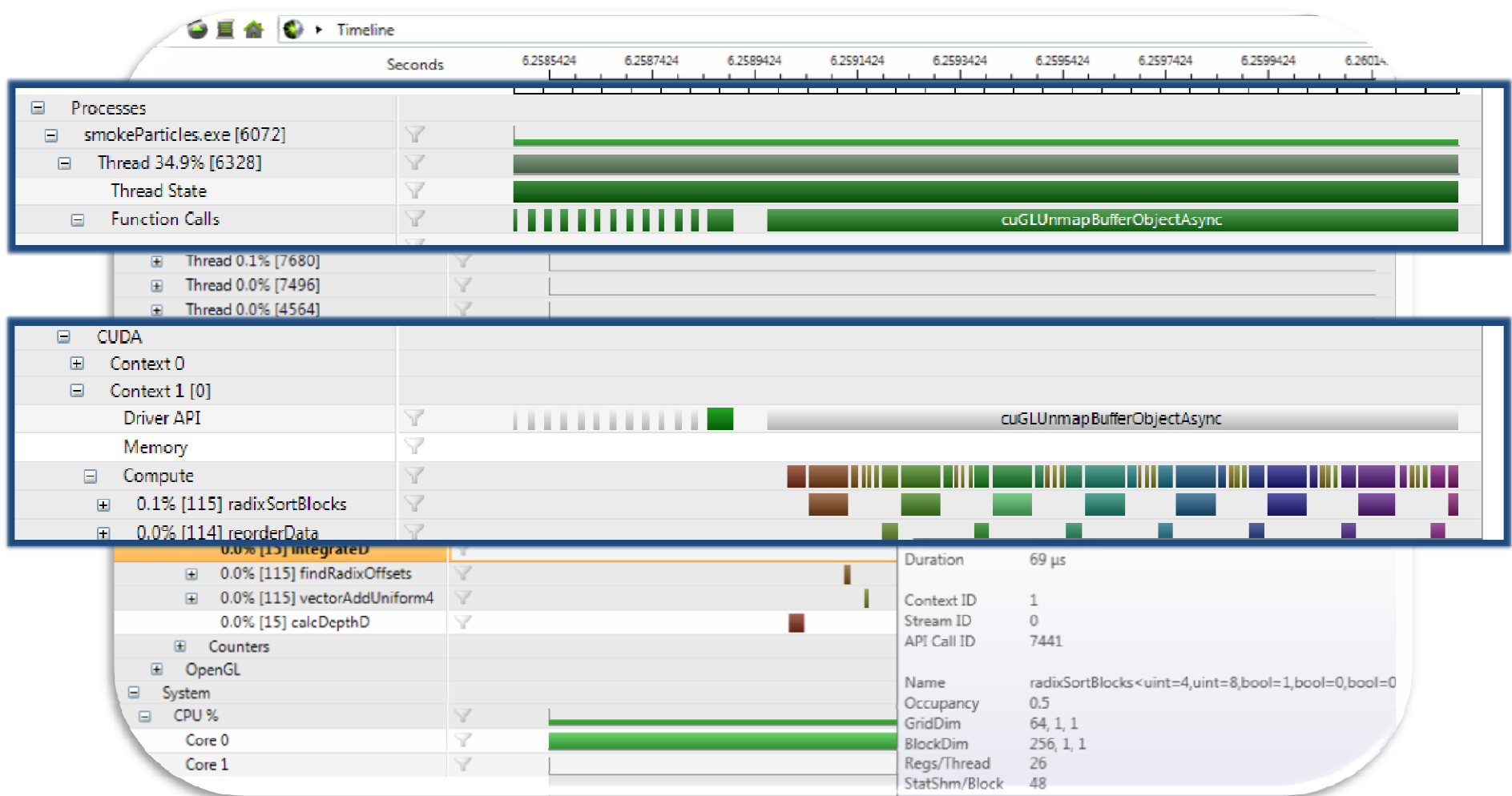


Nsight Analysis: Correlated Trace Plot





Nsight Analysis: Correlated Trace Plot





Tool Summary

Tool	Type	Availability
CUDA-GDB	Debugger	CUDA toolkit (linux only)
Visual Profiler	Profiler	CUDA toolkit (all OS)
CUDA-MemCheck	Memory Debugger	Beta, CUDA toolkit (all OS)
Parallel Nsight, MSVS integration	IDE (build/debug/profile)	Beta (Win7 or Vista)
TotalView Debugger	Debugger	
Allinea Distributed Debugging Tool	Debugger	
CUDA-GDB, Eclipse Integration by Fixstars on YDEL	IDE (build/debug)	



Instructional Resources

- **CUDA toolkit:**
 - Programming Guides, Best Practices Guides, compiler/profiler documentation
- **Programming basics:**
 - CUDA C:
 - http://developer.download.nvidia.com/CUDA/training/NVIDIA_GPU_Computing_Webinars_Introduction_to_CUDA. {wmv,pdf}
 - {wmv, pdf} = {video, slides}
 - OpenCL: GTC session **1409**
- **Advanced programming (optimization):**
 - CUDA C:
 - GTC session **1029** (slides and video)
 - CUDA Tutorials at Supercomputing: <http://gpgpu.org/{sc2007,sc2008,sc2009}>
 - OpenCL: GTC session **1068**
 - Kernel optimization is identical for CUDA C and OpenCL
- **GTC Materials for all sessions:**
 - <http://developer.download.nvidia.com/compute/cuda/docs/GTC09Materials.htm>