2017 KIAS CAC HPC summer school

CONTENTS

GPU Architecture

The first GPUs were designed as graphics accelerators, supporting only specific fixed-function pipelines. Starting in the late 1990s, the hardware became increasingly programmable, culminating in NVIDIA's first general purpose GPU in 1999. Less than a year after NVIDIA coined the term GPU, artists and game developers weren't the only ones doing ground-breaking work with the technology: Researchers were tapping its excellent floating point performance. The General Purpose GPU (GPGPU) movement had dawned. But GPGPU programming was far from easy back then, even for those who knew graphics programming languages such as OpenGL. Developers had to map scientific calculations onto problems that could be represented by triangles and polygons. GP-GPU programming was practically off-limits to those who hadn't memorized the latest graphics APIs until a group of Stanford University researchers set out to reimagine the GPU as a "streaming processor." In 2003, a team of researchers led by Ian Buck unveiled Brook, the first widely adopted programming model to extend the C programming language with data-parallel constructs. Using concepts such as streams, kernels and reduction operators, the Brook compiler and runtime system exposed the GPU as a general-purpose processor in a high-level language. Most importantly, Brook programs were not only easier to write than hand-tuned GPU code, they were seven times faster than similar existing code. NVIDIA knew that blazingly fast hardware had to be coupled with intuitive software and hardware tools, and invited Ian Buck to join the company and start evolving a solution to seamlessly run C on the GPU. Putting the software and hardware together, NVIDIA unveiled CUDA in 2006, the world's first solution for general-computing on GPUs.

Kepler GK110 was built first and foremost for Tesla, and its goal was to be the highest performing parallel computing microprocessor in the world. GK110 not only greatly exceeds the raw compute horsepower delivered by Fermi, but it does so efficiently, consuming significantly less power and generating much less heat output.

A full Kepler GK110 implementation includes 15 SMX units and six 64-bit memory controllers. Different products will use different configurations of GK110. For example, some products may deploy 13 or 14 SMXs. A principal design goal for the Kepler architecture was improving power efficiency. When designing Kepler, NVIDIA engineers applied everything learned from Fermi to better optimize the Kepler architecture for highly efficient operation. TSMC’s 28nm manufacturing process plays an important role in lowering power consumption, but many GPU architecture modifications were required to further reduce power consumption while maintaining great performance.

Every hardware unit in Kepler was designed and scrubbed to provide outstanding performance per watt. The best example of great perf/watt is seen in the design of Kepler GK110’s new Streaming Multiprocessor (SMX), which is similar in many respects to the SMX unit recently introduced in Kepler GK104, but includes substantially more double precision units for compute algorithms.

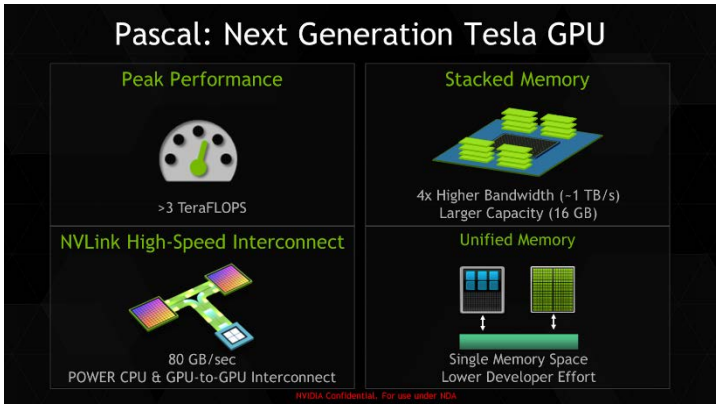
In 2014, NVIDIA launched Kepler GK210 based on GK110. There are some improvements in GK210. The below is the comparison table of Kepler GPUs.

Product	K40	K80
GPU	single GK110B	2x GK210
CUDA cores(SP)	2880	3992 (2496 per GPU)
Peak double Precision (GFLOPS)	1.43	1.87
Peak single Precision (TFLOPS)	4.29	5.6

Memory Bandwidth (GB/s)	288	480
Power (W)	235	300
Memory Size (GB)	12	24
PCI-e Connectivity	PCIe Gen3	PCIe Gen3

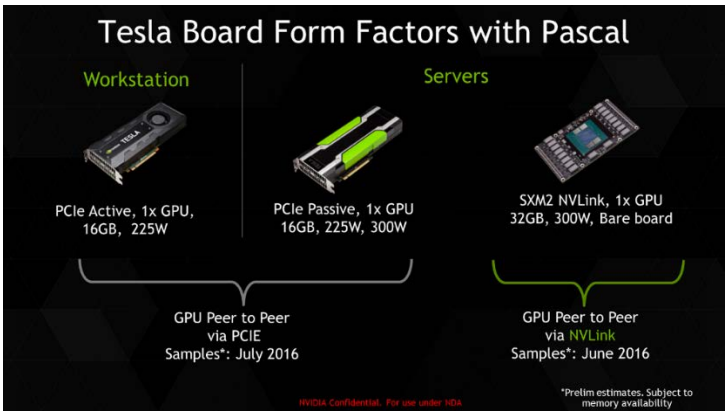
NVIDIA GPU roadmap

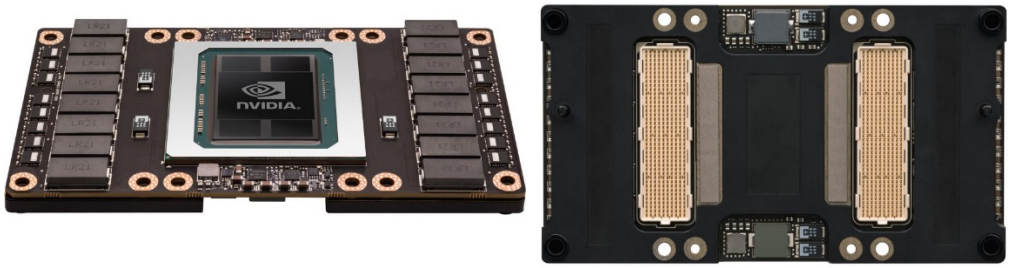
During a keynote speech at NVIDIA's annual GPU Technology Conference in San Jose, Calif., NVIDIA CEO Jen-Hsun Huang updated our public GPU roadmap with the announcement of Pascal, the GPU family that will follow Maxwell GPUs. Named for 17th century French mathematician Blaise Pascal, our next-generation family of GPUs will include three key new features: stacked DRAM, unified memory, and NVLink.



- **3D Memory:** Stacks DRAM chips into dense modules with wide interfaces, and brings them inside the same package as the GPU. This lets GPUs get data from memory more quickly – boosting throughput and efficiency – allowing us to build more compact GPUs that put more power into smaller devices. The result: several times greater bandwidth, more than twice the memory capacity and quadrupled energy efficiency.
- **Unified Memory:** This will make building applications that take advantage of what both GPUs and CPUs can do quicker and easier by allowing the CPU to access the GPU's memory, and the GPU to access the CPU's memory, so developers don't have to allocate resources between the two.
- **NVLink:** Today's computers are constrained by the speed at which data can move between the CPU and GPU. NVLink puts a fatter pipe between the CPU and GPU, allowing data to flow at more than 80GB per second, compared to the 16GB per second available now.
- **Pascal Module:** NVIDIA has designed a module to house Pascal GPUs with NVLink. At one-third the size of the standard boards used today, they'll put the power of GPUs into more compact form factors than ever before.

NVIDIA Tesla Board Form Factors



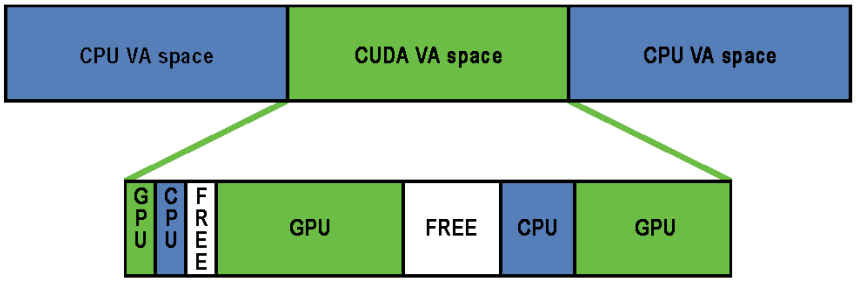


Tesla P100 SMX2

	Tesla P100	Tesla K80	Tesla K40	Tesla M40
Architecture	Pascal	Kepler	Kepler	Maxwell
GPU	GP100	GK210	GK110B	GM200
Stream Processors	3584	2 x 2496	2880	3072
Core Clock	1328MHz	562MHz	745MHz	948MHz
Boost Clock(s)	1480MHz	875MHz	875MHz	1114MHz
Memory	HBM2	GDDR5	GDDR5	GDDR5
Memory Clock	1.4Gbps	5GHz	6GHz	6GHz
Memory Bus Width	4096-bit	2 x 384-bit	384-bit	384-bit
Memory Bandwidth	720GB/sec	2 x 240GB/sec	288GB/sec	288GB/sec
VRAM	16GB	2 x 12GB	12GB	12GB
Half Precision	21.2 TFLOPS	8.74 TFLOPS	4.29 TFLOPS	6.8 TFLOPS
Single Precision	10.6 TFLOPS	8.74 TFLOPS	4.29 TFLOPS	6.8 TFLOPS
Double Precision	5.3 TFLOPS	2.91 TFLOPS	1.43 TFLOPS	213 GFLOPS
DP ratio	(1/2 rate)	(1/3 rate)	(1/3 rate)	(1/32 rate)
TDP	300W	300W	235W	250W
Cooling	N/A	Passive	Active/Passive	Passive
Manufacturing Process	TSMC 16nm FinFET	TSMC 28nm	TSMC 28nm	TSMC 28nm

Unified Virtual Addressing(UVA)

Unified virtual addressing (UVA) is a memory address management system enabled by default in CUDA 4.0 and later releases on Fermi and Kepler GPUs running 64-bit processes. The design of UVA memory management provides a basis for the operation of GPUDirect RDMA. On UVA-supported configurations, when the CUDA runtime initializes, the virtual address (VA) range of the application is partitioned into two areas: the CUDA-managed VA range and the OS-managed VA range. All CUDA-managed pointers are within this VA range, and the range will always fall within the first 40 bits of the process's VA space.



Subsequently, within the CUDA VA space, addresses can be subdivided into three types:

GPU : A page backed by GPU memory. This will not be accessible from the host and the VA in question will never have a physical backing on the host. Dereferencing a pointer to a GPU VA from the CPU will trigger a segfault.

CPU : A page backed by CPU memory. This will be accessible from both the host and the GPU at the same VA.

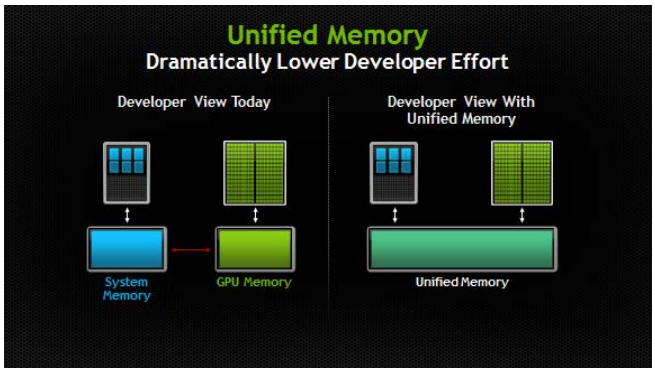
FREE : These VAs are reserved by CUDA for future allocations.

This partitioning allows the CUDA runtime to determine the physical location of a memory object by its pointer value within the reserved CUDA VA space.

Addresses are subdivided into these categories at page granularity; all memory within a page is of the same type. Note that GPU pages may not be the same size as CPU pages. The CPU pages are usually 4KB and the GPU pages on Kepler-class GPUs are 64KB. GPUDirect RDMA operates exclusively on GPU pages (created by cudaMalloc()) that are within this CUDA VA space.

Unified Memory (UM)

Unified Memory creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that the system automatically *migrates* data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU.



SIMPLER PROGRAMMING AND MEMORY MODEL

Unified Memory lowers the bar of entry to parallel programming on the CUDA platform, by making device memory management an optimization, rather than a requirement. With Unified Memory, now programmers can get straight to developing parallel CUDA kernels without getting bogged down in details of allocating and copying device memory. This will make both learning to program for the CUDA platform and porting existing code to the GPU simpler.

PERFORMANCE THROUGH DATA LOCALITY

By migrating data on demand between the CPU and GPU, Unified Memory can offer the performance of local data on the GPU, while providing the ease of use of globally shared data. The complexity of this functionality is kept under the covers of the CUDA driver and runtime, ensuring that application code is simpler to write. The point of migration is to achieve full bandwidth from each processor; the 250 GB/s of GDDR5 memory is vital to feeding the compute throughput of a Kepler GPU.

An important point is that a carefully tuned CUDA program that uses streams and cudaMemcpyAsync to efficiently overlap execution with data transfers may very well perform better than a CUDA program that only uses Unified Memory. the CUDA runtime never has as much information as the programmer does about where data is needed and when CUDA programmers still have access to explicit device memory allocation and asynchronous memory copies to optimize data management and CPU-GPU concurrency. Unified Memory is first and foremost a productivity feature that provides a smoother on-ramp to parallel computing, without taking away any of CUDA's features for power users.

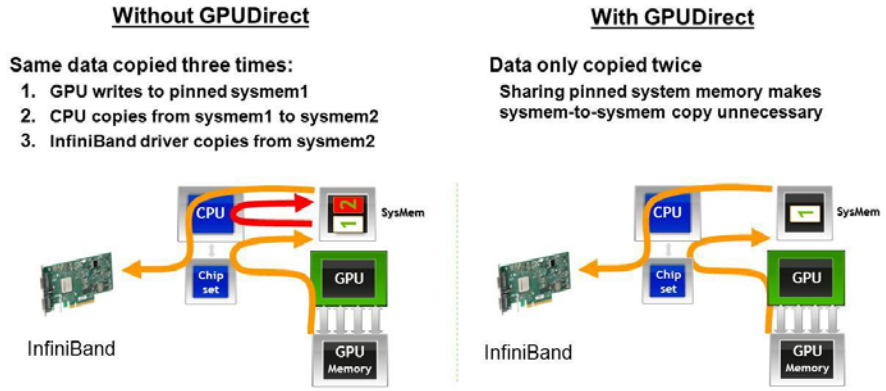
GPUDirect

Using GPUDirect, multiple GPUs, third party network adapters, solid-state drives (SSDs) and other devices can directly read and write CUDA host and device memory, eliminating unnecessary memory copies, dramatically lowering CPU overhead, and reducing latency, resulting in significant performance improvements in data transfer times for applications running on NVIDIA Tesla™ and Quadro™ products

GPUDirect includes a family of technologies that is continuously being evolved to increase performance and expand its usability. First introduced in June 2010, GPUDirect Shared Access supports accelerated communication with third party PCI Express device drivers via shared pinned host memory. In 2011, the release of GPUDirect Peer to Peer added support for Transfers and direct load and store Access between GPUs on the same PCI Express root complex. Announced in 2013, GPU Direct RDMA enables third party PCI Express devices to directly access GPU bypassing CPU host memory altogether.

GPUDirect Shared Access

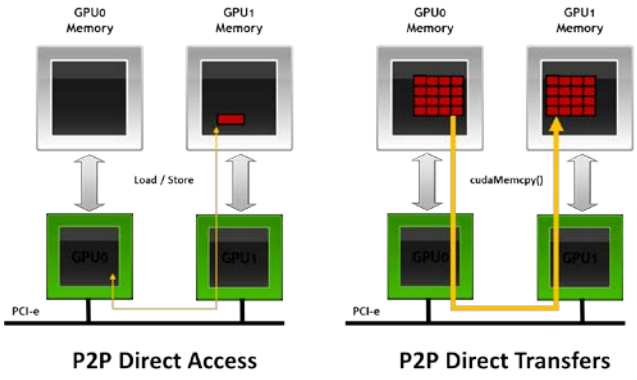
GPUDirect Shared Access supports accelerated communication with third party PCI Express device drivers via shared pinned host memory.



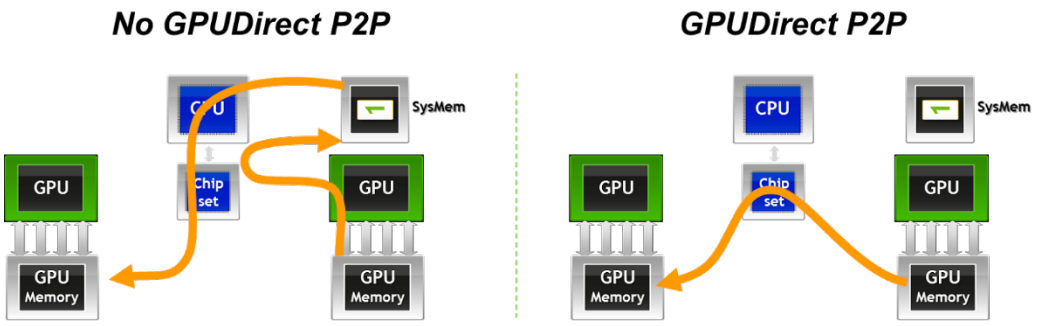
Host memory allocated with malloc is usually pageable, that is, the memory pages associated with the memory can be moved around by the kernel, for example to the swap partition on the hard drive. Memory paging has an impact on copying data by DMA and RDMA. DMA and RDMA transfers work independently of the CPU and thus also independently of the OS kernel, so memory pages must not be moved by the kernel while they are being copied. Inhibiting the movement of memory pages is called memory "pinning". So memory that cannot be moved by the kernel and thus can be used in DMA and RDMA transfers is called pinned memory. As a side note, pinned memory can also be used to speed up host-to-device and device-to-host transfer in general.

GPUDirect P2P

GPUDirect peer-to-peer transfers and memory access are supported natively by the CUDA Driver. All you need is CUDA Toolkit v4.0 and R270 drivers (or later) and a system with two or more Fermi- or Kepler-architecture GPUs on the same PCIe bus. For more information on using GPUDirect communication in your applications, please see:



which was introduced with CUDA 4.0 and can accelerate intra-node communication. Buffers can be directly copied between the memories of two GPUs in the same system with GPUDirect P2P.



GPUDirect RDMA

GPUDirect RDMA is a technology introduced in Kepler-class GPUs and CUDA 5.0 that enables a direct path for data exchange between the GPU and a third-party peer device using standard features of PCI Express. Examples of third-party devices are: network interfaces, video acquisition devices, storage adapters.

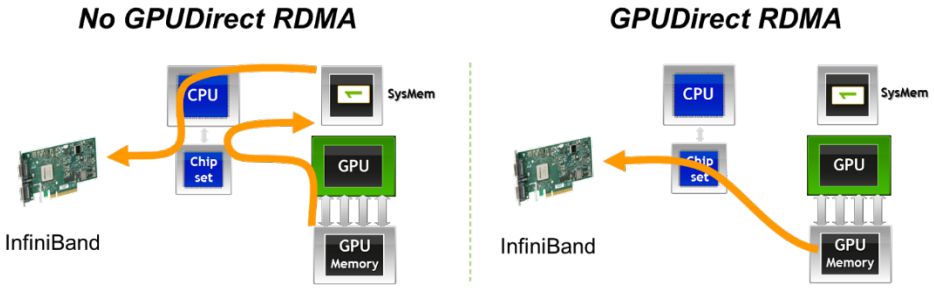
GPUDirect RDMA is available on both Tesla and Quadro GPUs.

A number of limitations can apply, the most important being that the two devices must share the same upstream PCI Express root complex. Some of the limitations depend on the platform used and could be lifted in current/future products.

A few straightforward changes must be made to device drivers to enable this functionality with a wide range of hardware devices. This document introduces the technology and describes the steps necessary to enable an GPUDirect RDMA connection to NVIDIA GPUs on Linux.

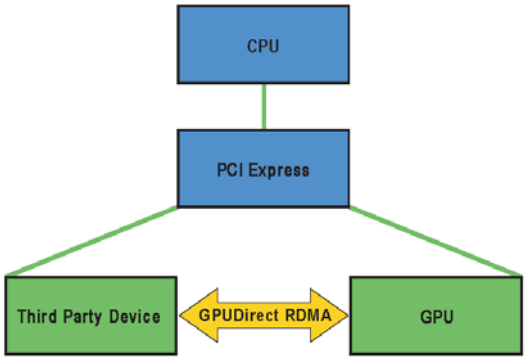
When setting up GPUDirect RDMA communication between two peers, all physical addresses are the same from the PCI Express devices' point of view. Within this physical address space are linear windows called PCI BARs. Each device has six BAR registers at most, so it can have up to six active 32bit BAR regions. 64 bit BARs consume two BAR registers. The PCI Express device issues reads and writes to a peer device's BAR addresses in the same way that they are issued to system memory.

Traditionally, resources like BAR windows are mapped to user or kernel address space using the CPU's MMU as memory mapped I/O (MMIO) addresses. However, because current operating systems don't have sufficient mechanisms for exchanging MMIO regions between drivers, the NVIDIA kernel driver exports functions to perform the necessary address translations and mappings.



To add GPUDirect RDMA support to a device driver, a small amount of address mapping code within the kernel driver must be modified. This code typically resides near existing calls to get_user_pages().

The APIs and control flow involved with GPUDirect RDMA are very similar to those used with standard DMA transfers.



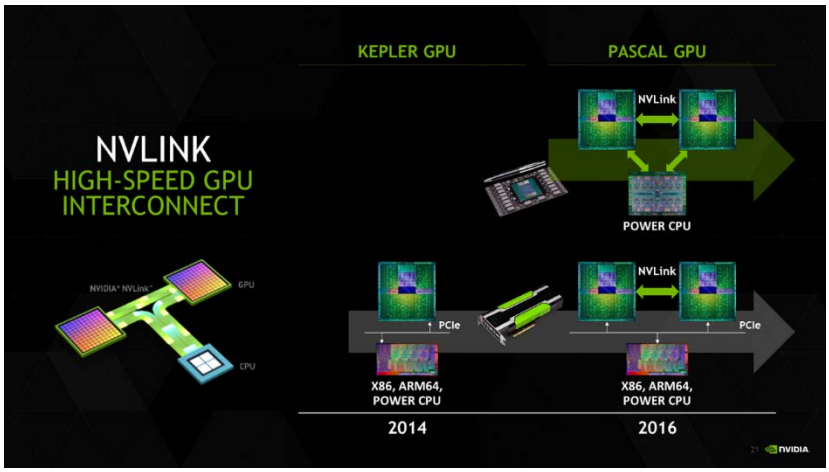
If GPUDirect RDMA is available the buffer can be directly moved to the network without touching the host memory at all. So the data is directly moved from the buffer in the device memory of MPI Rank 0 to the device memory of MPI Rank 1 with a PCI-E DMA → RDMA → PCI-E DMA sequence. If MPI rank 0 and MPI rank 1 are running on the same host and using GPUs on the same PCI-E bus GPUDirect P2P can be utilized to achieve a similar result.

If no variant of GPUDirect is available, for example if the network adapter does not support GPUDirect, the situation is a little bit more complicated. The buffer needs to be first moved to the pinned CUDA driver buffer and from there to the pinned buffer of the network fabric in the host memory of MPI Rank 0. After that it can be sent over the network. On the receiving MPI Rank 1 these steps need to be carried out in reverse.

NVLINK

NVIDIA® NVLink™ is a high-bandwidth, energy-efficient interconnect that enables ultra-fast communication between the CPU and GPU, and between GPUs. The technology allows data sharing at rates 5 to 12 times faster than the traditional PCIe Gen3 interconnect, resulting in dramatic speed-ups in application performance and creating a new breed of high-density, flexible servers for accelerated computing. Download the whitepaper for more details on NVLink. - See more at: http://www.nvidia.com/object/nvlink.html#sthash.iV56YFOa.dpuf

Designed to solve the challenges of exascale computing, NVLink is a fundamental ingredient of the U.S. Department of Energy's next-generation supercomputers. These new systems—"Summit" at Oak Ridge National Laboratory and "Sierra" at Lawrence Livermore National Laboratory—will be three times faster than today's fastest supercomputer.



The Importance of Heterogeneous Nodes

ORNL and LLNL chose to build the *Summit* and *Sierra* pre-exascale systems around this powerful heterogeneous compute model using technologies from IBM and NVIDIA. IBM's POWER CPUs are among the world's fastest serial processors. NVIDIA GPU accelerators are the most efficient general purpose throughput-oriented processors on the planet. Coupling them together produces a highly efficient and optimized heterogeneous node capable of minimizing both serial and parallel sections of HPC codes.The architectural emphasis on parallelism in GPUs leads to optimization for throughput, hiding rather than minimizing latency. Support for thousands of threads ensures a ready pool of work in the face of data dependencies in order to sustain performance at a high percent of peak. The memory hierarchy design and technology thoroughly reflect optimization for throughput performance at minimal energy per bit.

By contrast, latency-optimized CPU architecture drives completely different design decisions. Techniques designed to compress the execution of a single instruction thread into the smallest possible time demand a host of architectural features (like branch prediction, speculative execution, register renaming) that would cost far too much energy to be replicated for thousands of parallel GPU threads but that are entirely appropriate for CPUs.

The essence of the heterogeneous computing model is that one size does not fit all. Parallel and serial segments of the workload execute on the best-suited processor—latency-optimized CPU or throughput-optimized GPU—delivering faster overall performance, greater efficiency, and lower energy and cost per unit of computation.

The Growing Multi-GPU Trend

Since *Titan*, a trend has emerged toward heterogeneous node configurations with larger ratios of GPU accelerators per CPU socket, with two or more GPUs per CPU becoming common as developers increasingly expose and leverage the available parallelism in their applications. Although each of the new DoE systems is unique, they share the same fundamental multi-GPU node architecture.

While multi-GPU applications provide a vehicle for scaling single node performance, they can be constrained by interconnect performance between the GPUs. Developers must overlap data transfers with computation or carefully orchestrate GPU accesses over PCIe interconnect to maximize

performance. However, as GPUs get faster and GPU-to-CPU ratios climb, a higher performance node integration interconnect is warranted. Enter NVLink.

NVLink: High-Speed GPU Interconnect

NVLink is an energy-efficient, high-bandwidth path between the GPU and the CPU at data rates of at least 80 gigabytes per second, or at least 5 times that of the current PCIe Gen3 x16, delivering faster application performance. NVLink is the node integration interconnect for both the *Summit* and *Sierra*pre-exascale supercomputers commissioned by the U.S. Department of Energy, enabling NVIDIA GPUs and CPUs such as IBM POWER to access each other's memory quickly and seamlessly. NVLink will first be available with the next-generation NVIDIA Pascal™ GPU in 2016.

In addition to speeding CPU-to-GPU communications for systems with an NVLink CPU connection, NVLink can have significant performance benefit for GPU-to-GPU (peer-to-peer) communications as well. A second new NVIDIA white paper focuses on these peer-to-peer benefits from NVLink**,** showing how systems with next-generation NVLink-interconnected GPUs are projected to deliver considerable application speedup compared to systems with GPUs interconnected via PCIe.

The white paper analyzes the performance benefit of NVLink for several algorithms and applications by comparing model systems based on PCIe-interconnected next-gen GPUs to otherwise-identical systems with NVLink-interconnected GPUs. GPUs are connected to the CPU using existing PCIe connections, but the NVLink configurations augment this with interconnections among the GPUs for peer-to-peer communication.

The paper examines five multi-GPU algorithms and applications important to HPC: exchange and sort, FFT, AMBER Molecular Dynamics (PMEMD), ANSYS Fluent Computational Fluid Dynamics (CFD), and QUDA Lattice Quantum Chromodynamics (LQCD). Projected results for exchange, sort, and 3D FFT are shown in Figures 1, 2, and 3, respectively.

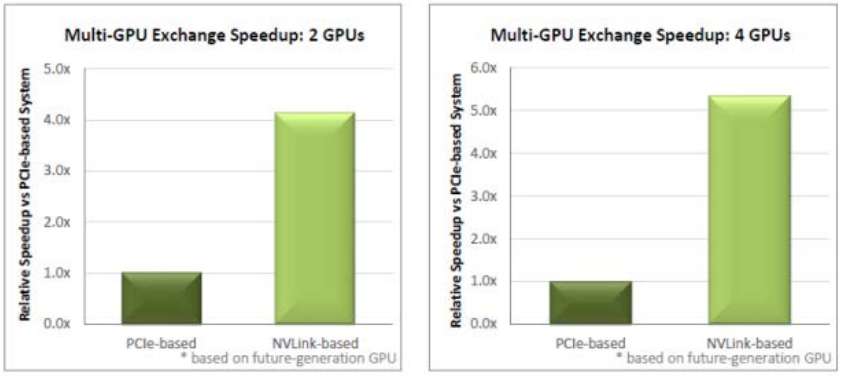


Figure 1: Projected multi-GPU exchange performance in 2-GPU and 4-GPU configurations, comparing NVLink-based systems to PCIe-based systems.

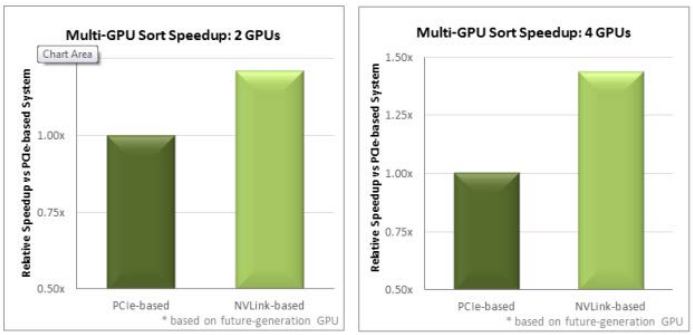


Figure 2: Projected multi-GPU sorting performance in 2-GPU and 4-GPU configurations, comparing NVLink-based systems to PCIe-based systems.

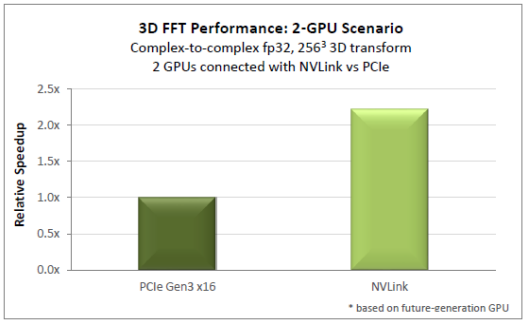


Figure 3: Projected 3D FFT performance in 2-GPU configurations. NVLink-connected GPUs deliver over 2x speedup.

The whitepaper presents the analysis of performance considerations for each application, with the result that NVLink is projected to deliver significant performance boost – up to 2x in many applications – simply by replacing the PCIe interconnect for communication among peer GPUs. This clearly illustrates the growing challenge NVLink addresses: as the GPU computation rate grows, GPU interconnect speeds must scale up accordingly in order to see the full benefit of the faster GPU.

NVLink is a flexible and scalable interconnect technology, enabling a rich set of design options for next-generation servers to include multiple GPUs with a variety of interconnect topologies and bandwidths, as Figure 4 shows.

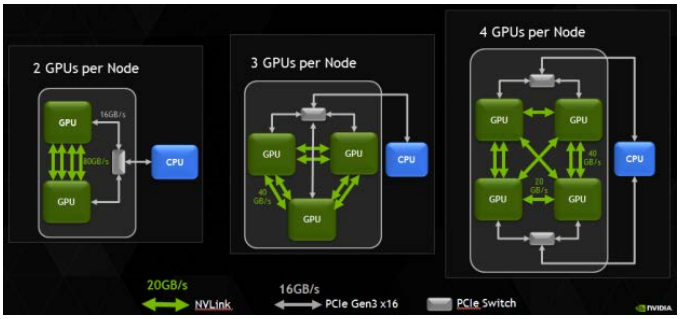


Figure 4: NVLink will enable flexible configuration of multiple GPU accelerators in next-generation servers.

Faster, Easier Programming

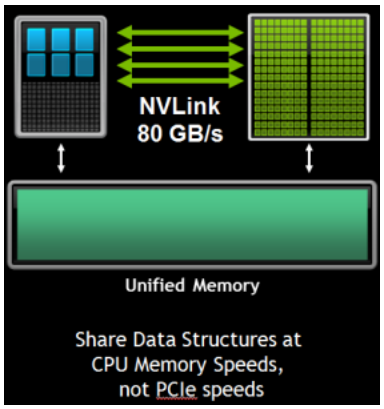
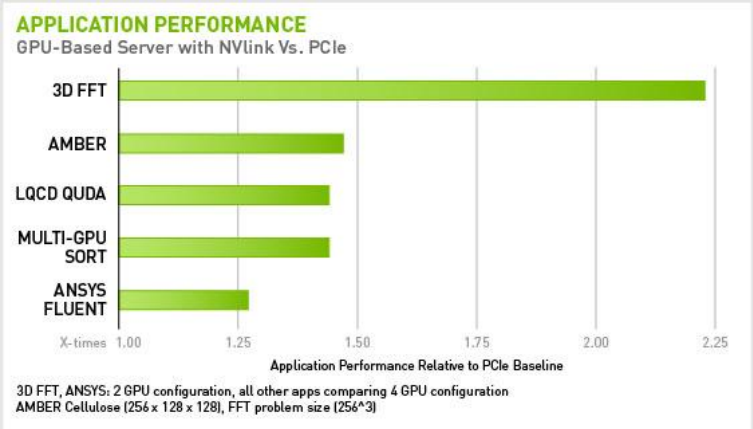


Figure 5: The combination of Unified Memory and NVLink will enable faster, easier data sharing between CPU and GPU code.

One important result of the higher bandwidth between GPUs provided by NVLink will be that libraries such as cuFFT and cuBLAS can offer much better multi-GPU scalability, scaling onto a greater number of GPUs as well as strong scaling smaller problems where communication is a significant bottleneck today.

Unified Memory and NVLink represent a powerful combination for CUDA® programmers. Unified Memory provides you with a single pointer to data and automatic migration of that data between the CPU and GPU. With 80 GB/s or higher bandwidth on machines with NVLink-connected CPUs and GPUs, that means GPU kernels will be able to access data in host system memory at the same bandwidth the CPU has to that memory—much faster than PCIe. Host and device portions of applications will be able to share data much more efficiently and cooperatively operate on shared data structure, and supporting larger problem sizes will be easier than ever.



Perf. Projection with NVLINK vs PCI-e

CUDA Dev. Environment

CUDA® is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the graphics processing unit (GPU).

CUDA was developed with several design goals in mind: •Provide a small set of extensions to standard programming languages, like C, that enable a straightforward implementation of parallel algorithms. With CUDA C/C++, programmers can focus on the task of parallelization of the algorithms rather than spending time on their implementation.
•Support heterogeneous computation where applications use both the CPU and GPU. Serial portions of applications are run on the CPU, and parallel portions are offloaded to the GPU. As such, CUDA can be incrementally applied to existing applications. The CPU and GPU are treated as separate devices that have their own memory spaces. This configuration also allows simultaneous computation on the CPU and GPU without contention for memory resources.
CUDA-capable GPUs have hundreds of cores that can collectively run thousands of computing threads. These cores have shared resources including a register file and a shared memory. The on-chip shared memory allows parallel tasks running on these cores to share data without sending it over the system memory bus.
This guide will show you how to install and check the correct operation of the CUDA development tools.
CUDA-capable GPUs have hundreds of cores that can collectively run thousands of computing threads. These cores have shared resources including a register file and a shared memory. The on-chip shared memory allows parallel tasks running on these cores to share data without sending it over the system memory bus.

NVCC Compiler

The CUDA Toolkit targets a class of applications whose control part runs as a process on a general purpose computing device, and which use one or more NVIDIA GPUs as coprocessors for accelerating single program, multiple data (SPMD) parallel jobs. Such jobs are self-contained, in the sense that they can be executed and completed by a batch of GPU threads entirely without intervention by the host process, thereby gaining optimal benefit from the parallel graphics hardware.

The GPU code is implemented as a collection of functions in a language that is essentially C++, but with some annotations for distinguishing them from the host code, plus annotations for distinguishing different types of data memory that exists on the GPU. Such functions may have parameters, and they can be called using a syntax that is very similar to regular C function calling, but slightly extended for being able to specify the matrix of GPU threads that must execute the called function. During its lifetime, the host process may dispatch many parallel GPU tasks.

For more information on the CUDA programming model, consult the CUDA C Programming Guide.

Source files for CUDA applications consist of a mixture of conventional C++ host code, plus GPU device functions. The CUDA compilation trajectory separates the device functions from the host code, compiles the device functions using the proprietary NVIDIA compilers and assembler, compiles the host code using a C++ host compiler that is available, and afterwards embeds the compiled GPU functions as fat binary images in the host object file. In the linking stage, specific CUDA runtime libraries are added for supporting remote SPMD procedure calling and for providing explicit GPU manipulation such as allocation of GPU memory buffers and host-GPU data transfer.

The compilation trajectory involves several splitting, compilation, preprocessing, and merging steps for each CUDA source file. It is the purpose of nvcc, the CUDA compiler driver, to hide the intricate details of CUDA compilation from developers. It accepts a range of conventional compiler options, such as for defining macros and include/library paths, and for steering the compilation process. All non-CUDA compilation steps are forwarded to a C++ host compiler that is supported by nvcc, and nvcc translates its options to appropriate host compiler command line options.

A general purpose C++ host compiler is needed by nvcc in the following situations:
• During non-CUDA phases (except the run phase), because these phases will be forwarded by nvcc to this compiler.
• During CUDA phases, for several preprocessing stages and host code compilation (see also The CUDA Compilation Trajectory).

nvcc assumes that the host compiler is installed with the standard method designed by the compiler provider. If the host compiler installation is non-standard, the user must make sure that the environment is set appropriately and use relevant nvcc compile options.

A compilation phase is the a logical translation step that can be selected by command line options to nvcc. A single compilation phase can still be broken up by nvcc into smaller steps, but these smaller steps are just implementations of the phase: they depend on seemingly arbitrary capabilities of the internal tools that nvcc uses, and all of these internals may change with a new release of the CUDA Toolkit. Hence, only compilation phases are stable across releases, and although nvcc provides

options to display the compilation steps that it executes, these are for debugging purposes only and must not be copied and used into build scripts.

nvcc phases are selected by a combination of command line options and input file name suffixes, and the execution of these phases may be modified by other command line options. In phase selection, the input file suffix defines the phase input, while the command line option defines the required output of the phase.

The following table defines how nvcc interprets its input files:

Input File Prefix	Description
.cu	CUDA source file, containing host code and device functions
.c	C source file
.cc, .cxx,	C++ source file
.gpu	GPU intermediate file
.ptx	PTX intermediate assembly file
.o, .obj	Object file
.a, .lib	Library file
.res	Resource file
.so	Shared object file

Note that nvcc does not make any distinction between object, library or resource files. It just passes files of these types to the linker when the linking phase is executed.

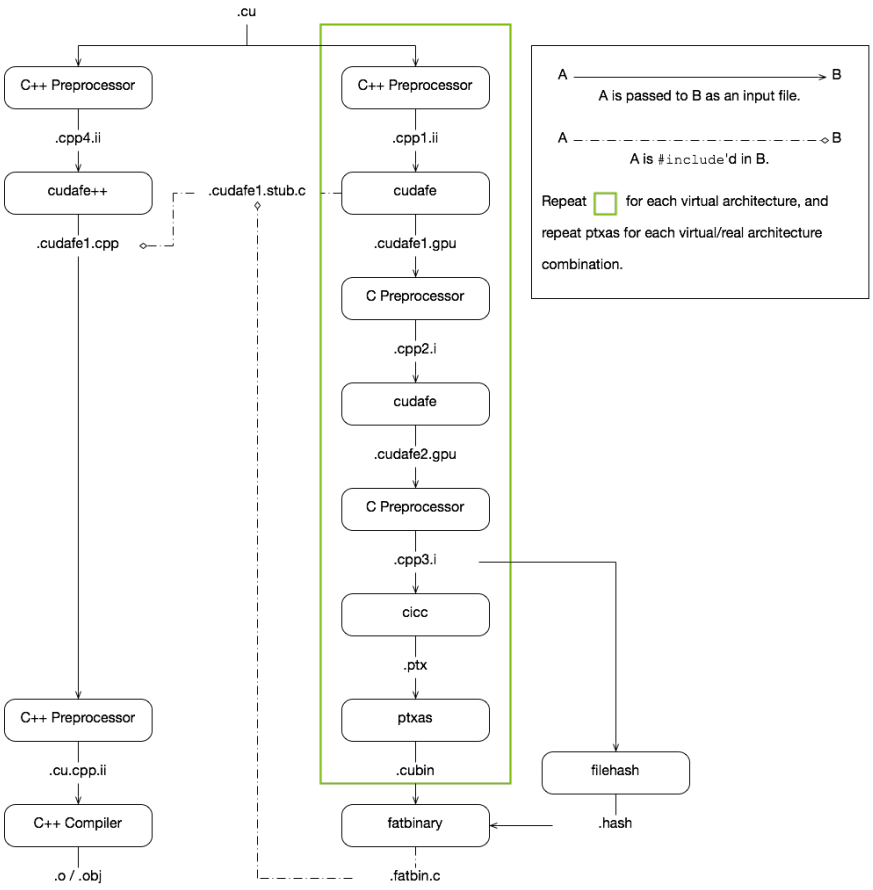
CUDA Compilation Trajectory

The CUDA phase converts a source file coded in the extended CUDA language into a regular ANSI C++ source file that can be handed over to a general purpose C++ host compiler for further compilation and linking. The exact steps that are followed to achieve this are displayed in Figure 1.

CUDA compilation works as follows: the input program is preprocessed for device compilation compilation and is compiled to CUDA binary (cubin) and/or PTX intermediate code, which are placed in a fat binary. The input program is preprocesed once again for host compilation and is synthesized to embed the fat binary and transform CUDA specific C++ extensions into standard C++ constructs. Then the C++ host compiler compiles the synthesized host code with the embedded fat binary into a host object.

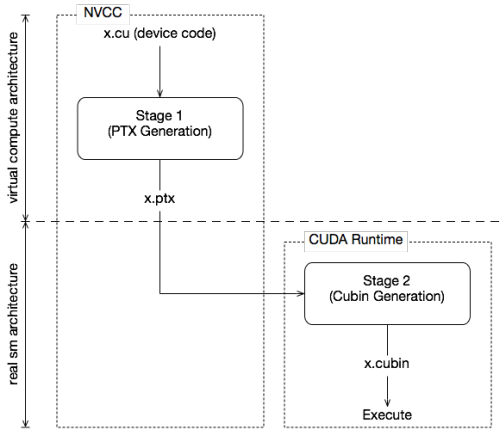
The embedded fat binary is inspected by the CUDA runtime system whenever the device code is launched by the host program to obtain an appropriate fat binary image for the current GPU.

The CUDA compilation trajectory is more complicated in the separate compilation mode.



CUDA Whole Program Compilation Trajectory

The compilation step to an actual GPU binds the code to one generation of GPUs. Within that generation, it involves a choice between GPU *coverage* and possible performance. For example, compiling to sm_30 allows the code to run on all Kepler-generation GPUs, but compiling to sm_35 would probably yield better code if Kepler GK110 and later are the only targets.



OpenACC compiler

OpenACC is an open specification for compiler directives for parallel programming. By using OpenACC, developers can rapidly accelerate existing C, C++, and Fortran applications using high-level directives that help retain application portability across processor architectures. Figure 1 shows some examples of real code speedups with OpenACC. The OpenACC specification is designed and maintained with the cooperation of many industry and academic partners, such as Cray, AMD, PathScale, University of Houston, Oak Ridge National Laboratory and NVIDIA.

OpenACC gives scientists and researchers a simple and powerful way to accelerate scientific computing without significant programming effort. The toolkit includes the PGI OpenACC Compiler, the NVIDIA Visual Profiler with CPU and GPU profiling, and the new OpenACC Programming and Best Practices Guide. Academics can get a free renewable license to the PGI C,C++ and Fortran compilers with support for OpenACC.

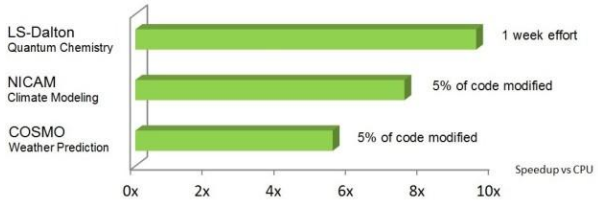


Figure 1: LS-DALTON: Benchmark on Oak Ridge Titan Supercomputer, AMD CPU vs Tesla K20X GPU. Test input: Alanine-3 on CCSD(T) module. Additional information: NICAM COSMO

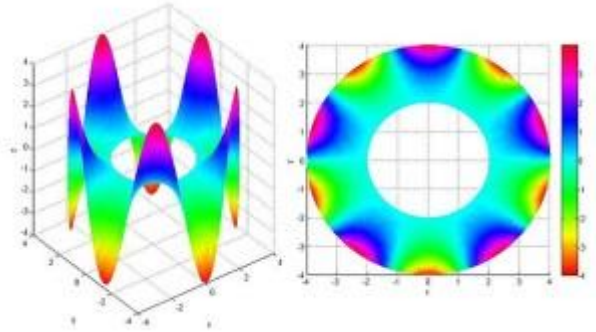
4 Step of OpenACC Development

4 step cycle to progressively accelerate the code.

1. **Identify Parallelism:** Profile the code to understand where the program is spending its time and how much parallelism is available to be accelerated in those important routines. GPUs excel when there's a significant amount of parallelism to exploit, so look for loops and loop nests with a lot of independent iterations.
2. **Express Parallelism:** Placing OpenACC directives on the loops identified in step 1 tells the compiler to parallelize them. OpenACC is all about giving the compiler enough information to effectively accelerate the code, so during this step I add directives to as many loops as I believe I can and move as much of the computation to the GPU as possible.
3. **Express Data Locality:** The compiler needs to know not just what code to parallelize, but also which data will be needed on the accelerator by that code. After expressing available parallelism, I often find that the code has slowed down. As you'll see later in this post, this slowdown comes from the compiler making cautious decisions about when data needs to be moved to the GPU for computation. During this step, I'll express to the compiler my knowledge of when and how the data is really needed on the GPU.
4. **Optimize** – The compiler usually does a very good job accelerating code, but sometimes you can get more performance by giving the compiler a little more information about the loops or by restructuring the code to increase parallelism or improve data access patterns. Most of the time this leads to small improvements, but sometimes gains can be bigger.

Case Study : Jacobi Iteration

The benchmark code is a simple C code that solves the 2D Laplace equation with the iterative Jacobi solver. Iterative methods are a common technique to approximate the solution of elliptic PDEs, like the Laplace equation, within an allowable tolerance. In the example, it is a simple stencil calculation where each point calculates its value as the mean of its neighbors' values. The Jacobi iteration continues until either the maximum change in value between two iterations drops below the tolerance level or it exceeds the maximum number of iterations. For the sake of consistent comparison all the results I show are for 1000 complete iterations. Here's the main Jacobi iteration loop.



```
          while ( error > tol && iter < iter_max ) {
           error = 0.0;
           for( int j = 1; j < n-1; j++) {
             for( int i = 1; i < m-1; i++ ) {
               Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1] +
                                     A[j-1][i] + A[j+1][i]);
               error = fmax( error, fabs(Anew[j][i] - A[j][i]));
             }
           }

           for( int j = 1; j < n-1; j++) {
             for( int i = 1; i < m-1; i++ ) {
               A[j][i] = Anew[j][i];
             }
           }

           if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
           iter++;
 }
```

The outer loop iterates for 1000 iterations or until the results converge to within an acceptable tolerance (whichever comes first). I'll refer to this loop as the *convergence loop*. The two j, i loop nests within the convergence loop do the main calculation. The first loop calculates the new values for each point in the matrix as the mean of the neighboring values. The second loop nest copies these values into the A array for the next iteration. Note that the code calculates an error value for each element, which is how much the value changed between iterations, and finds the maximum value oferror to determine whether the solution has converged.

In this code it's clear that the convergence loop cannot be parallelized because of the dependence of each iteration on the results of the previous. This is known as a *data dependency*. The two inner loop nests over i and j, however, can be parallelized, as each iteration writes its own unique value and does not read from the values calculated in other iterations. These loops can be executed forward, backward, in a random order, or all simultaneously and the results will be the same (modulo floating point error). Therefore, the two loop nests are candidates for acceleration. Developer will use the OpenACC kernels directive to accelerate them. The kernels directive tells the compiler to analyze the code in the specified region to look for parallel loops. In the following code, developer have placed a kernels region within the convergence loop, around the two loop nests that perform the calculation and value swapping, since this is where the compiler is most likely to find parallelism.

```
          while ( error > tol && iter < iter_max ) {
           error = 0.0;
           #pragma acc kernels
           {
             for( int j = 1; j < n-1; j++) {
               for( int i = 1; i < m-1; i++ ) {
                 Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                     + A[j-1][i] + A[j+1][i]);
                 error = fmax( error, fabs(A[j][i] - Anew[j][i]));
               }
             }

             for( int j = 1; j < n-1; j++) {
               for( int i = 1; i < m-1; i++ ) {
                 A[j][i] = Anew[j][i];
               }
             }
           }

           if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
           iter++;
 }
```

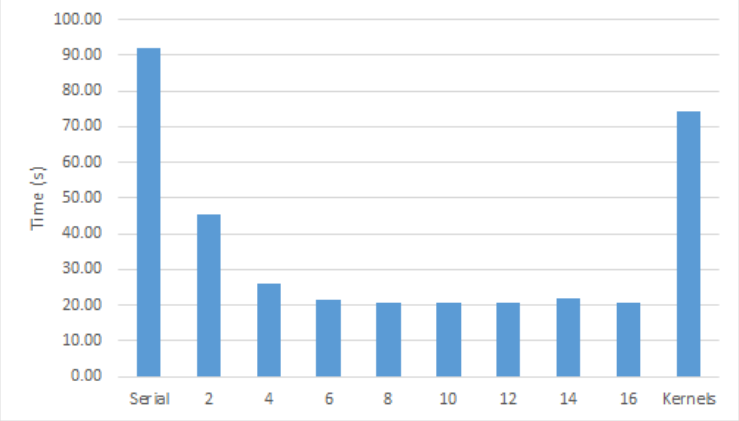
Having inserted an OpenACC directive developer now want to build the code. For this developer'll use the PGI C/C++ compiler, pgcc, which is included in the OpenACC Toolkit. Developer want the compiler to generate code specific to NVIDIA GPUs, so developer add the -ta=tesla compiler option (ta means *target accelerator* and tesla targets NVIDIA Tesla GPUs). Developer also use the optional -Minfo=accel compiler flag to tell the compiler to provide feedback about the code it generates. Below is the output from this command.

```
          $ pgcc -acc -ta=tesla -Minfo=accel laplace2d-kernels.c
          main:
              56, Generating copyout(Anew[1:4094][1:4094])
                  Generating copyin(A[:][:])
                  Generating copyout(A[1:4094][1:4094])
                  Generating Tesla code
              58, Loop is parallelizable
              60, Loop is parallelizable
                  Accelerator kernel generated
                  58, #pragma acc loop gang /* blockIdx.y */
                  60, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
                  64, Max reduction generated for error
              68, Loop is parallelizable
              70, Loop is parallelizable
```

```
                Accelerator kernel generated
                68, #pragma acc loop gang /* blockIdx.y */
        70, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
```

Notice from the compiler output that the compiler identified two regions of code for generating an accelerator kernel. (A kernel is simply a function that is run in parallel on the accelerator / GPU). In this case, the loop nests starting at lines 58 and 68 have been identified as safe to parallelize and the compiler has generated kernels for these loops. The compiler also analyzed which arrays are used in the calculation and generated code to move A and Anew into GPU memory. The compiler even detected that it needs to perform a *max reduction* on the error variable.

A reduction is an operation that takes many input values and combines them into a single result using a binary operator. In this c+ase, each loop iteration calculates an error value, but the iteration logic only needs one result: the maximum error. Reductions can be tricky in some parallel programming languages, but fortunately the OpenACC compiler handled the reduction for me. Now that I've built the code, user can expect it to run significantly faster on GPU, right? Figure 3 shows the performance of this code running on multiple cores of an Intel Xeon E5-2698 CPU to performance on an NVIDIA Tesla K40 GPU. The CPU code was parallelized using OpenMP and compiled with PGI 15.5 and -fast optimizations.




```
        #pragma acc data copy(A) create(Anew)
        while ( error > tol && iter < iter_max )
        {
         error = 0.0;

         #pragma acc kernels
         {
           for( int j = 1; j < n-1; j++) {
             for( int i = 1; i < m-1; i++ ) {
               Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                   + A[j-1][i] + A[j+1][i]);
               error = fmax( error, fabs(A[j][i] - Anew[j][i]));
             }
           }

           for( int j = 1; j < n-1; j++) {
             for( int i = 1; i < m-1; i++ ) {
               A[j][i] = Anew[j][i];
             }
```

```
            }
          }

          if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
          iter++;
}
```

If developer added a data region around the while loop, since this is where developer can reuse the data on the GPU. Developer used the copy clause to tell the compiler that it should copy the A array to and from the device as it enters and exits the region, respectively. Since the Anew array is only used within the convergence loop, developer tell the compiler to create temporary space on the device, since developer don't care about the initial or final values of that array. There are other data clauses that don't use in this example:

- copyin initializes the value of the array but does not copy the results back;
- copyout allocates uninitialized space on the device and copies the final values back; and
- present tells the compiler to assume that the array was moved or allocated on the device somewhere else in the program.

This single-line data clause makes a big difference to performance. Figure 5 shows that with data locality properly expressed the code runs six times faster than a full CPU socket with almost no data motion.

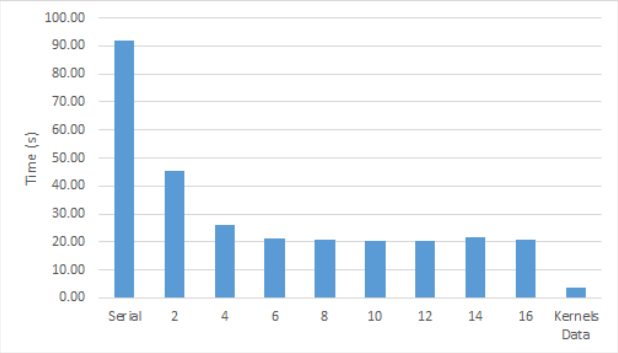


Figure 6 shows the NVIDIA Visual Profiler timeline, which shows that there's no data motion between convergence iterations, just as expected.

OpenACC Optimization

https://github.com/uhhpctools/openacc-npb

NPB in OpenACC which is based on SNU NPB suite.

SNU NPB Suite is a set of the NAS Parallel Benchmarks (NPB) implemented in C, OpenMP C, and OpenCL. Current SNU NPB Suite consists of four different implementations: NPB-SER-C, NPB-OMP-C, NPB-OCL, NPB-OCL-MD.

In NPB, most of the benchmarks contain many global arrays live throughout the entire program. To speed up, need to allocate memory at the beginning and update directive to synchronize data between host and device. Moreover, asyncdirective to overlap communication and computation increase the speed. To accelerate Accelerate NPB in OpenACC, there are multiple optimization techniques :

- Array privatization
- Loop scheduling tuning
- Memory coalescing optimization
- Scalar substitution
- Loop fission and unrolling
- Data motion optimization

In the simple example show how to optimize the base-line code with Array privatization, loop scheduling tuning, memory coalescing optimization, scalar substitution instead of array and loop unrolling. OPT5 source code describe how to speed up with loop unrolling technique.

Optimization Steps	code

Original CPU code
Baseline
 	
BT/exact_rhs.c
	`for(k=0; k<=grid_points[2]-1; k++){`
	`  for(j=0; j<grid_points[1]-1; j++){`
	`  for(i=0; i<grid_points[0]-1; i++){`
	`  for(m=0; m<5; m++){`
	`  rhs[j][i][m] = forcing[k][j][i][m];`
	`  }`
	`  }`
	`  }`
	`}`

OpenACC
Baseline
Directives
	`#pragma acc kernels`
	`for(k=0; k<=grid_points[2]-1; k++){`
	`  for(j=0; j<grid_points[1]-1; j++){`
	`  for(i=0; i<grid_points[0]-1; i++){`
	`  for(m=0; m<5; m++){`
	`  rhs[j][i][m] = forcing[k][j][i][m];`
	`  }`
	`  }`
	`  }`
	`}`

OpenACC
OPT1
Vector privatization
	`#pragma acc kernels`
	`for(k=0; k<=grid_points[2]-1; k++){`
	`  for(j=0; j<grid_points[1]-1; j++){`
	`  for(i=0; i<grid_points[0]-1; i++){`
	`  for(m=0; m<5; m++){`
	`  rhs[k][j][i][m]= forcing[k][j][i][m];`
	`  }`
	`  }`
	`  }`
	`}`

OpenACC
OPT2
	`#pragma acc kernels loop gang`
	`for(k=0; k<=grid_points[2]-1; k++){`
	`#pragma acc loop worker`
	`  for(j=0; j<grid_points[1]-1; j++){`
	`#pragma acc loop vector`

Control loop schedule for independent loop	`for(i=0; i<grid_points[0]-1; i++){`
	`for(m=0; m<5; m++){`
	`rhs[k][j][i][m] = forcing[k][j][i][m];`
	`}`
	`}`
	`}`
	`}`

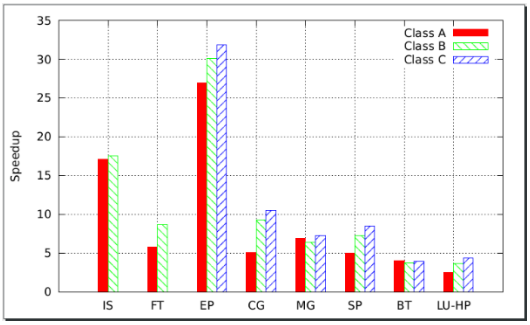
OpenACC
OPT3
Memory access
	`#pragma acc kernels loop gang`
	`for(k=0; k<=grid_points[2]-1; k++){`
	`#pragma acc loop worker`
	`for(j=0; j<grid_points[1]-1; j++){`
	`#pragma acc loop vector`
	`for(i=0; i<grid_points[0]-1; i++){`
	`for(m=0; m<5; m++){`
	`rhs[m][k][j][i] = forcing[m][k][j][i];`
	`}`
	`}`
	`}`
	`}`

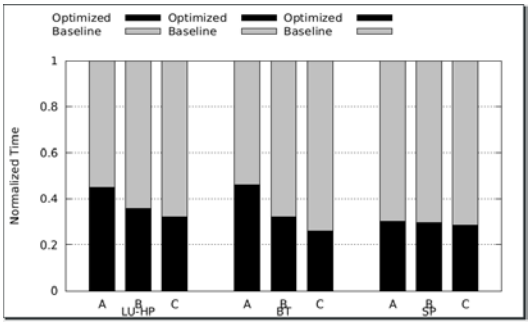
OpenACC
OPT4
Scalar substitution
	`#pragma acc kernels loop gang`
	`for(k=0; k<=gp2-1; k++){`
	`#pragma acc loop worker`
	`for(j=0; j<gp1-1; j++){`
	`#pragma acc loop vector`
	`for(i=0; i<gp0-1; i++){`
	`for(m=0; m<5; m++){`
	`rhs[m][k][j][i] = forcing[m][k][j][i];`
	`}`
	`}`
	`}`
	`}`

OpenACC
OPT5
Loop unroll
	`#pragma acc kernels loop gang`
	`for(k=0; k<=gp2-1; k++){`
	`#pragma acc loop worker`
	`for(j=0; j<gp1-1; j++){`
	`#pragma acc loop vector`
	`for(i=0; i<gp0-1; i++){`
	`rhs[0][k][j][i] = forcing[0][k][j][i];`
	`rhs[1][k][j][i] = forcing[1][k][j][i];`
	`rhs[2][k][j][i] = forcing[2][k][j][i];`
	`rhs[3][k][j][i] = forcing[3][k][j][i];`
	`rhs[4][k][j][i] = forcing[4][k][j][i];`
	`}`
	`}`

Performance evaluation on Kepler Architecture.

The chart show the perf. For NPB in OpenACC on Tesla K20. SP achieve 5~10X speed up and BT achieve 4X speed up. Regarding the problem size, SP get 2X speed up in Class C rather and Class A. Both of BT and SP, NPB in OpenACC got more 2X speed up with optimization.


Speedup
35
30
25
20
15
10
5
0
Class A
Class B
Class C
IS
FT
EP
CG
MG
SP
BT
LU-HP


Optimized
Baseline
Optimized
Baseline
Optimized
Baseline
Normalized Time
1
0.8
0.6
0.4
0.2
0
A
B
C
LU-HP
A
B
C
BT
A
B
C
SP

Development Tools

CUDA-GDB

CUDA-GDB is the NVIDIA tool for debugging CUDA applications running on Linux and Mac. CUDA-GDB is an extension to the x86-64 port of GDB, the GNU Project debugger. The tool provides developers with a mechanism for debugging CUDA applications running on actual hardware. This enables developers to debug applications without the potential variations introduced by simulation and emulation environments.

CUDA-GDB is designed to present the user with a seamless debugging environment that allows simultaneous debugging of both GPU and CPU code within the same application. Just as programming in CUDA C is an extension to C programming, debugging with CUDA-GDB is a natural extension to debugging with GDB. The existing GDB debugging features are inherently present for debugging the host code, and additional features have been provided to support debugging CUDA device code.

CUDA-GDB supports debugging C/C++ and Fortran CUDA applications. (Fortran debugging support is limited to 64-bit Linux operating system) All the C++ features supported by the NVCC compiler can be debugged by CUDA-GDB.

CUDA-GDB allows the user to set breakpoints, to single-step CUDA applications, and also to inspect and modify the memory and variables of any given thread running on the hardware.

CUDA-GDB supports debugging all CUDA applications, whether they use the CUDA driver API, the CUDA runtime API, or both.

CUDA-GDB supports debugging kernels that have been compiled for specific CUDA architectures, such as `sm_20` or `sm_30`, but also supports debugging kernels compiled at runtime, referred to as just-in-time compilation, or JIT compilation for short.

Usage of CUDA-GDB

To inspect the current focus, use the cuda command followed by the coordinates of interest:

```
(cuda-gdb) cuda device sm warp lane block thread
block (0,0,0), thread (0,0,0), device 0, sm 0, warp 0, lane 0
(cuda-gdb) cuda kernel block thread
kernel 1, block (0,0,0), thread (0,0,0)
(cuda-gdb) cuda kernel
kernel 1
```

To switch the current focus, use the cuda command followed by the coordinates to be changed:

```
(cuda-gdb) cuda thread (15)
[Switching focus to CUDA kernel 1, grid 2, block (8,0,0), thread (15,0,0), device 0, sm
1, warp 0, lane 15]
374 int totalThreads = gridDim.x * blockDim.x;
```

```
(cuda-gdb) cuda block 1 thread 3
[Switching focus to CUDA kernel 1, grid 2, block (1,0,0), thread (3,0,0), device 0, sm
3, warp 0, lane 3]
374 int totalThreads = gridDim.x * blockDim.
```

If the specified focus is not fully defined by the command, the debugger will assume that the omitted coordinates are set to the coordinates in the current focus, including the subcoordinates of the block and thread.

```
(cuda-gdb) print &array
$1 = (@shared int (*)[0]) 0x20
(cuda-gdb) print array[0]@4
$2 = {0, 128, 64, 192}
You can also access the shared
```

GNU DDD integration with CUDA-GDB

GNU DDD is a graphical front-end for command-line debuggers such as GDB, DBX, WDB, Ladebug, JDB, XDB, the Perl debugger, the bash debugger bashdb, the GNU Make debugger remake, or the Python debugger pydb. Besides ``usual'' front-end features such as viewing source texts, DDD has become famous through its interactive graphical data display, where data structures are displayed as graphs.

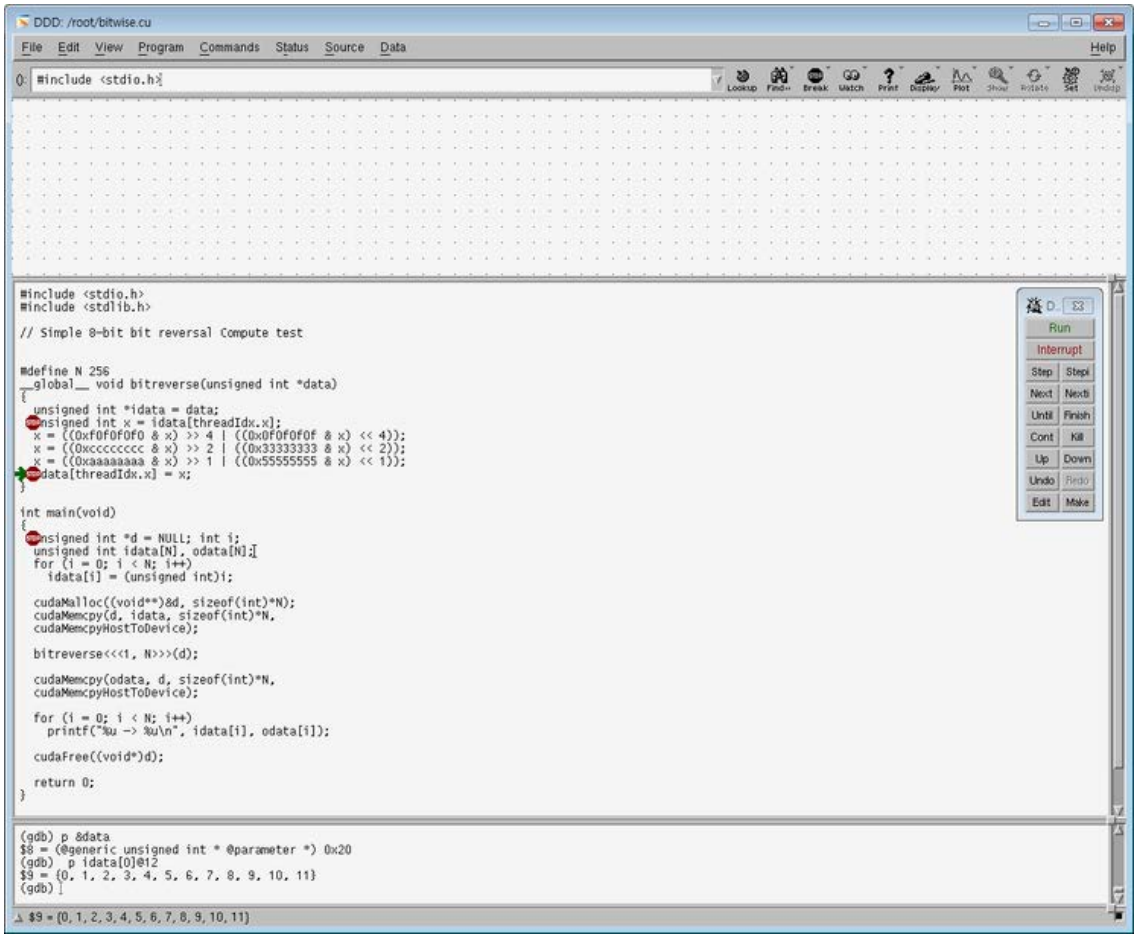


Figure – ddd integration with cuda-gdb

NSIGHT – GUI GPU Debugger

NVIDIA® Nsight™ Eclipse Edition is a unified CPU plus GPU integrated development environment (IDE) for developing CUDA® applications on Linux and Mac OS X for the x86, POWER and ARM platforms. It is designed to help developers on all stages of the software development process. Nsight Eclipse Edition is bundled in the NVIDIA CUDA Toolkit, so installing the CUDA Toolkit also installs Nsight. The principal features are as follows:

- Source editor with extended support for CUDA C and C++ syntax
- Projects and files management with version control management system integration. CVS and Git are supported out of the box with integrations for other systems available separately as IDE plug-ins.
- Configurable makefile-based NVCC build integration
- Graphical user interface for debugging heterogeneous applications
- Visual profiler with source code correlation for optimizing GPU code performance

Nsight Eclipse Edition is based on the popular Eclipse Platform and supports a wide range of the third-party extensions and plug-ins including:

- Version control management systems support
- Compiler integrations
- Language IDEs
- Application lifecycle management and collaboration solutions

For more information about Eclipse Platform, visit http://eclipse.org

Creating a new project in NSIGHT Eclipse Edition

a. From the main menu, open the new project wizard - **File** > **New...** > **CUDA C/C++ Project**
b. Specify the project name and project files location.
c. Select **CUDA Runtime Project** to create a simple CUDA runtime application.
d. Specify the project parameters on the next wizard page.
e. **Note:** By default Nsight will automatically detect and target CUDA hardware available locally. Nsight will default to SM 2.0 if no CUDA hardware is detected.
f. Complete the wizard. The project will be shown in the **Project Explorer** view and source editor will be opened.

g. Build the project by clicking on the hammer button on the main toolbar.

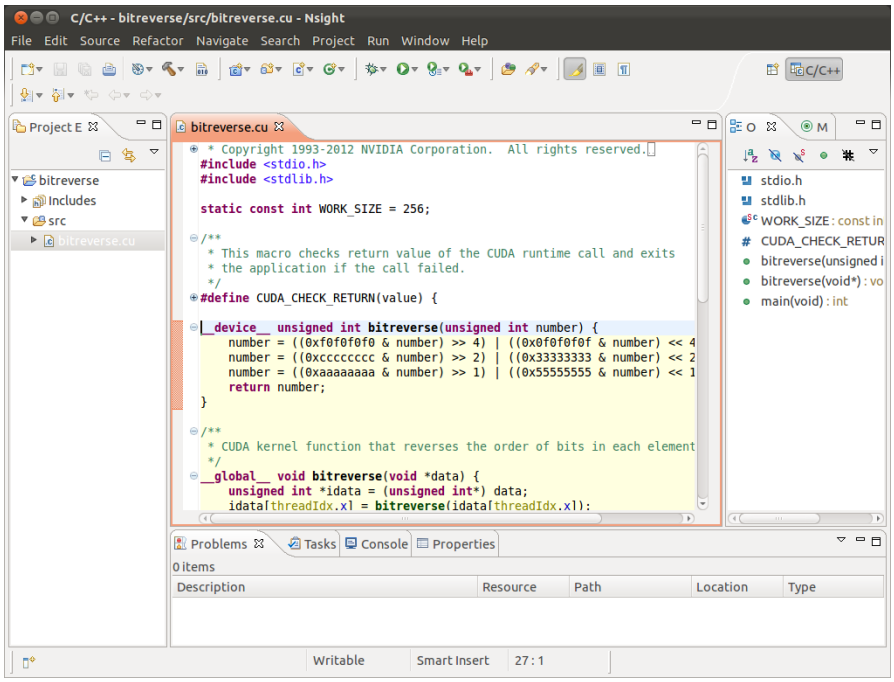


Figure - Nsight main window after creating a new project

Debugging on Nsight

a. In the **Project Explorer** view, select project you want to debug. Make sure the project executable is compiled and no error markers are shown on the project.

b. On the main window toolbar press **Debug** button (green bug).

c. You will be offered to switch perspective when you run debugger for the first time. Click "Yes". Perspective is a window layout preset specifically designed for a particular task.

d. Application will suspend in the *main* function. At this point there is no GPU code running.

e. Add a breakpoint in the device code. Resume the application.

Debugger will break when application reaches the breakpoint. You can now explore your CUDA device state, step through your GPU code or resume the application.

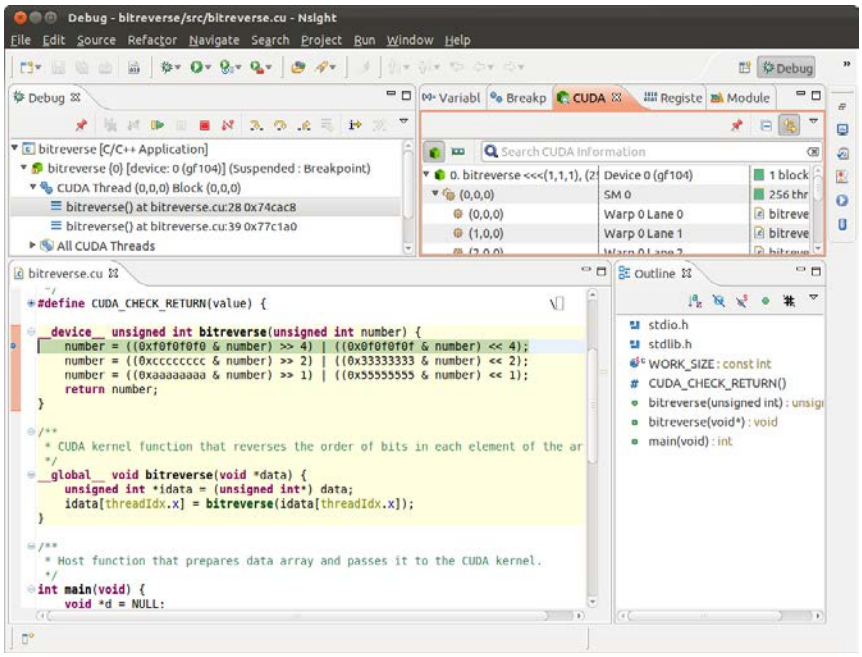


Figure - Debugging CUDA application on NSIGHT Eclipse Edition

CUDA-MEMCHECK

CUDA-MEMCHECK is a functional correctness checking suite included in the CUDA toolkit. This suite contains multiple tools that can perform different types of checks. The *memcheck* tool is capable of precisely detecting and attributing out of bounds and misaligned memory access errors in CUDA applications. The tool also reports hardware exceptions encountered by the GPU. The *racecheck* tool can report shared memory data access hazards that can cause data races.This document describes the usage of these tools. The *initcheck* tool can report cases where the GPU performs uninitialized accesses to global memory. The *synccheck* tool can report cases where the application is attempting to use CTA synchronization in divergent code.

CUDA-MEMCHECK can be run in *standalone mode* where the user's application is started under CUDA-MEMCHECK. The *memcheck* tool can also be enabled in *integrated mode* inside CUDA-GDB.

NVIDIA allows developers to easily harness the power of GPUs to solve problems in parallel using CUDA. CUDA applications often run thousands of threads in parallel. Every programmer invariably encounters memory access errors and thread ordering errors that are hard to detect and time consuming to debug. The number of such errors increases substantially when dealing with thousands of threads. The CUDA-MEMCHECK suite is designed to detect such errors in your CUDA application. Using the memcheck tool, CUDA-MEMCHECK can identify memory access errors as well as hardware reported program errors. The racecheck tool in CUDA-MEMCHECK can identify hazards caused by race conditions in the CUDA program.

Profiler

NVPROF

The nvprof profiling tool enables you to collect and view profiling data from the command-line. nvprof enables the collection of a timeline of CUDA-related activities on both CPU and GPU, including kernel execution, memory transfers, memory set and CUDA API calls. nvprof also enables you to collect events/metrics for CUDA kernels. Profiling options are provided to nvprof through command-line options. Profiling results are displayed in the console after the profiling data is collected, and may also be saved for later viewing by either nvprof or the Visual Profiler.

Note: The textual output is redirected to stderr by default. Use --log-file to redirect the output to another file. See Redirecting Output.

nvprof is included in the CUDA Toolkit for all supported OSes. Here's how to use nvprof to profile a CUDA application:

nvprof [options] [CUDA-application] [application-arguments]

Summary mode is the default operating mode for nvprof. In this mode, nvprof outputs a single result line for each kernel function and each type of CUDA memory copy/set performed by the application. For each kernel, nvprof outputs the total time of all instances of the kernel or type of memory copy as well as the average, minimum, and maximum time. The time for a kernel is the kernel execution time on the device. By default, nvprof also prints a summary of all the CUDA runtime/driver API calls. Output of nvprof (except for tables) are prefixed with ==<pid>==, <pid> being the process ID of the application being profiled. Here's a simple example of running nvprof on the CUDA sample matrixMul:

```
$ nvprof ./matrixMul
[Matrix Multiply Using CUDA] - Starting...
==20089== NVPROF is profiling process 20089, command: ./matrixMul
GPU Device 0: "Tesla K40m" with compute capability 3.5

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 350.82 GFlop/s, Time= 0.374 msec, Size= 131072000 Ops, WorkgroupSize= 1024
threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost
is enabled.
==20089== Profiling application: ./matrixMul
==20089== Profiling result:
Time(%)      Time     Calls       Avg       Min       Max  Name
99.81%  111.29ms       301  369.73us  366.47us  374.50us  void matrixMulCUDA<int=32>(float*,
float*, float*, int, int)
 0.11%  126.43us         2  63.216us  44.928us  81.505us  [CUDA memcpy HtoD]
 0.08%  85.281us         1  85.281us  85.281us  85.281us  [CUDA memcpy DtoH]

==20089== API calls:
Time(%)      Time     Calls       Avg       Min       Max  Name
57.44%  312.32ms         3  104.11ms  206.28us  311.90ms  cudaMalloc
20.26%  110.14ms         1  110.14ms  110.14ms  110.14ms  cudaDeviceReset
20.03%  108.89ms         1  108.89ms  108.89ms  108.89ms  cudaEventSynchronize
 0.90%  4.9132ms       498  9.8650us     125ns  387.65us  cuDeviceGetAttribute
```

```
 0.46%  2.4802ms       301  8.2390us  7.5720us  37.088us  cudaLaunch
 0.30%  1.6150ms         3  538.33us  299.64us  739.73us  cudaMemcpy
 0.15%  828.11us         1  828.11us  828.11us  828.11us  cudaGetDeviceProperties
 0.14%  777.54us         3  259.18us  236.64us  285.08us  cudaFree
 0.10%  531.60us         6  88.600us  81.872us  92.282us  cuDeviceTotalMem
 0.08%  455.00us         6  75.832us  71.584us  81.562us  cuDeviceGetName
 0.07%  396.69us         1  396.69us  396.69us  396.69us  cudaDeviceSynchronize
 0.05%  253.92us      1505     168ns     149ns  1.0430us  cudaSetupArgument
 0.02%  84.697us       301     281ns     259ns  2.3280us  cudaConfigureCall
 0.00%  10.724us         1  10.724us  10.724us  10.724us  cudaGetDevice
 0.00%  7.3690us         2  3.6840us  2.9050us  4.4640us  cudaEventRecord
 0.00%  6.7470us         2  3.3730us     852ns  5.8950us  cudaEventCreate
 0.00%  3.2380us         1  3.2380us  3.2380us  3.2380us  cudaEventElapsedTime
 0.00%  2.8700us        12     239ns     134ns     372ns  cuDeviceGet
 0.00%  2.2870us         2  1.1430us     371ns  1.9160us  cuDeviceGetCount
```

Table – nvprof summary result for matrixmul binary

GPU-Trace and API-Trace modes can be enabled individually or at the same time. GPU-trace mode provides a timeline of all activities taking place on the GPU in chronological order. Each kernel execution and memory copy/set instance is shown in the output. For each kernel or memory copy detailed information such as kernel parameters, shared memory usage and memory transfer throughput are shown. The number shown in the square brackets after the kernel name correlates to the CUDA API that launched that kernel.

```
$ nvprof --print-gpu-trace  ./matrixMul
[Matrix Multiply Using CUDA] - Starting...
==20285== NVPROF is profiling process 20285, command: ./matrixMul
GPU Device 0: "Tesla K40m" with compute capability 3.5

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 350.60 GFlop/s, Time= 0.374 msec, Size= 131072000 Ops, WorkgroupSize= 1024
threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost
is enabled.
==20285== Profiling application: ./matrixMul
==20285== Profiling result:
  Start  Duration            Grid Size      Block
Size     Regs*    SSMem*    DSMem*      Size  Throughput           Device   Context    Stream  Name
408.59ms  43.297us                    -               -         -         -         -  400.00KB  8.
8105GB/s   Tesla K40m (0)         1         7  [CUDA memcpy HtoD]
408.87ms  81.249us                    -               -         -         -         -  800.00KB  9.
3901GB/s   Tesla K40m (0)         1         7  [CUDA memcpy HtoD]
408.96ms  368.23us            (20 10 1)       (32 32
1)        29  8.0000KB        0B         -           -   Tesla K40m (0)         1         7  void
matrixMulCUDA<int=32>(float*, float*, float*, int, int) [538]
409.37ms  371.78us            (20 10 1)       (32 32
1)        29  8.0000KB        0B         -           -   Tesla K40m (0)         1         7  void
matrixMulCUDA<int=32>(float*, float*, float*, int, int) [549]
409.75ms  368.77us            (20 10 1)       (32 32
1)        29  8.0000KB        0B         -           -   Tesla K40m (0)         1         7  void
matrixMulCUDA<int=32>(float*, float*, float*, int, int) [556]
410.12ms  370.47us            (20 10 1)       (32 32
1)        29  8.0000KB        0B         -           -   Tesla K40m (0)         1         7  void
matrixMulCUDA<int=32>(float*, float*, float*, int, int) [563]
410.49ms  371.40us            (20 10 1)       (32 32
1)        29  8.0000KB        0B         -           -   Tesla K40m (0)         1         7  void
matrixMulCUDA<int=32>(float*, float*, float*, int, int) [570]
```

```
     -----   more output -------

521.14ms  369.92us            (20 10 1)       (32 32
1)        29  8.0000KB        0B         -           -   Tesla K40m (0)         1         7  void
matrixMulCUDA<int=32>(float*, float*, float*, int, int) [2642]
521.57ms  82.273us                     -              -         -         -         -  800.00KB  9.
2733GB/s   Tesla K40m (0)         1         7  [CUDA memcpy DtoH]

Regs: Number of registers used per CUDA thread. This number includes registers used internally by
the CUDA driver and/or tools and can be more than what the compiler shows.
SSMem: Static shared memory allocated per CUDA block.
DSMem: Dynamic shared memory allocated per CUDA block.
```

table – gpu api trace view for matrixmul binary

NVVP

The NVIDIA Visual Profiler allows you to visualize and optimize the performance of your CUDA application. The Visual Profiler displays a timeline of your application's activity on both the CPU and GPU so that you can identify opportunities for performance improvement. In addition, the Visual Profiler will analyze your application to detect potential performance bottlenecks and direct you on how to take action to eliminate or reduce those bottlenecks.

The Visual Profiler is available as both a standalone application and as part of Nsight Eclipse Edition. The standalone version of the Visual Profiler, nvvp, is included in the CUDA Toolkit for all supported OSes. Within Nsight Eclipse Edition, the Visual Profiler is located in the Profile Perspective and is activated when an application is run in profile mode. Nsight Eclipse Edition, nsight, is included in the CUDA Toolkit for Linux and Mac OSX.

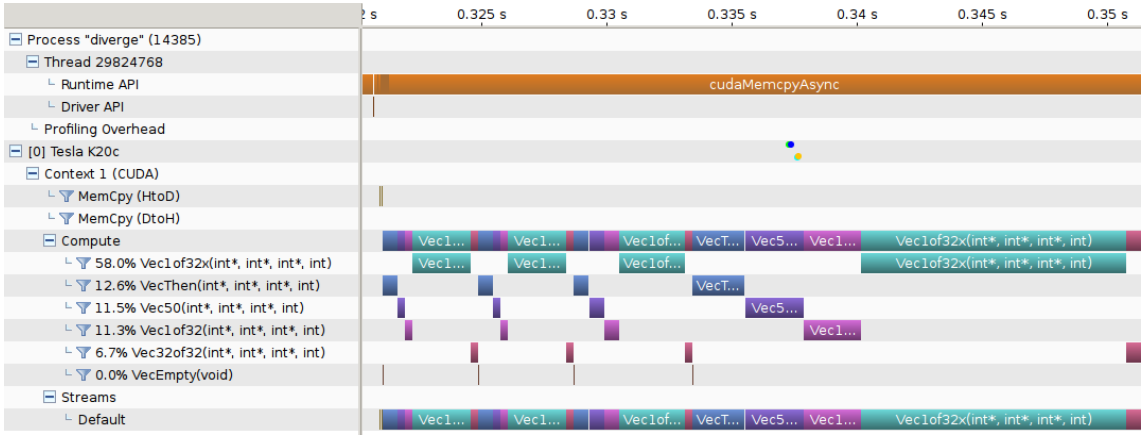
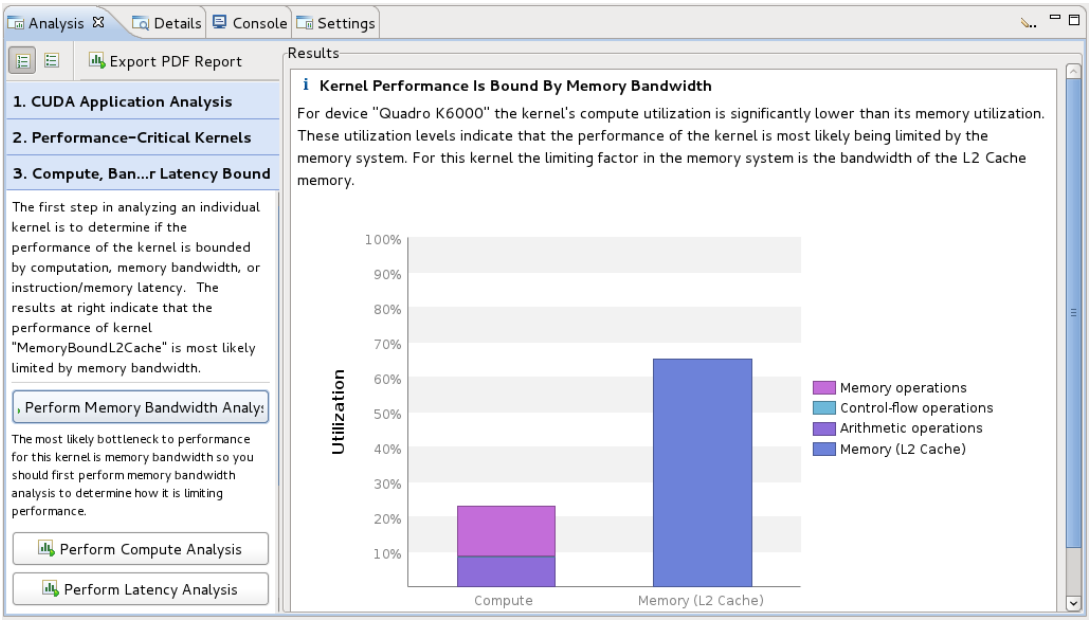


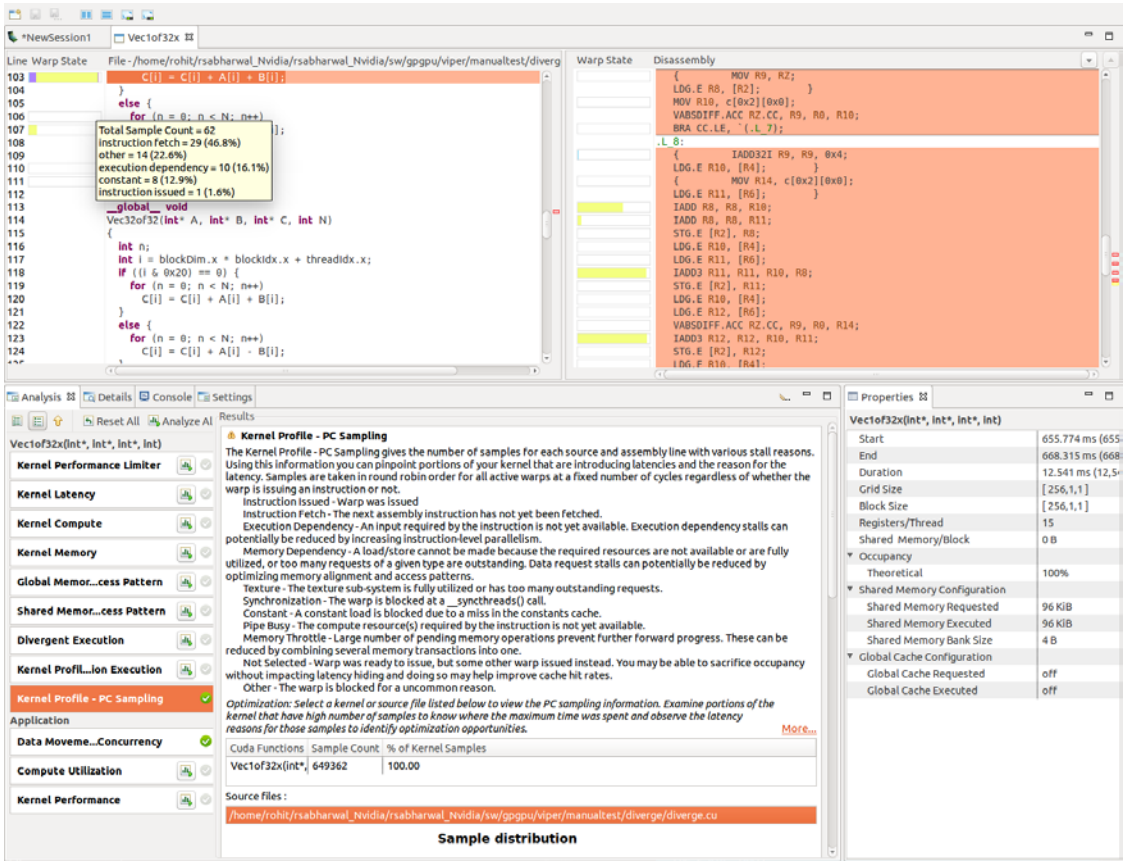
Figure – Timeline view in NVVP

The Analysis View is used to control application analysis and to display the analysis results. There are two analysis modes: *guided* and *unguided*. In guided mode the analysis system will guide you through multiple analysis stages to help you understand the likely performance limiters and optimization opportunities in your application. In unguided mode you can manually explore all the analysis results collect for you application. The following figure shows the analysis view in guided analysis mode. The left part of the view provides step-by-step directions to help you analyse and optimize your application. The right part of the view shows you detailed analysis results appropriate for each part of the analysis.



Devices with compute capability 5.2 have a feature for PC sampling. In this feature PC and state of warp is sampled at regular interval for one of the active warps per SM. The warp state indicates if the warp issued in that cycle or the stall reason why it could not issue in that cycle. When a warp that is sampled has a stall reason, there is a possibility that in the same cycle some other warp is issuing an instruction. Hence the stall for the sampled warp need not necessarily indicate that there is a hole in instruction issue pipeline. Refer the Warp State section for a description of different states.

The Visual Profiler collects this information and presents it in the **Kernel Profile - PC Sampling** view. In this view, in the **Results** sample distribution for all functions called from the kernel is given in the table. Also a pie chart is shown that gives the distribution of stall reasons collected for the kernel using sampling. After clicking on the source file or device function the **Kernel Profile - PC Sampling View** is opened. The hotspots shown next to the vertical scroll bar are decided based on number of samples collected for each source and assembly line. The distribution of the stall reasons is shown as a stacked bar for each source and assembly line. This helps in pinpointing the latency reasons at the source level.



Occupancy Calculator

CUDA GPU Occupancy Calculator

Click Here for detailed instructions on how to use this occup

For more information on NVIDIA CUDA, visit http://developer.nv

Just follow steps 1, 2, and 3 below! (or click here for help)

1.) Select Compute Capability (click):	3.5	(Help)
1.b) Select Shared Memory Size Config (bytes)	49152	

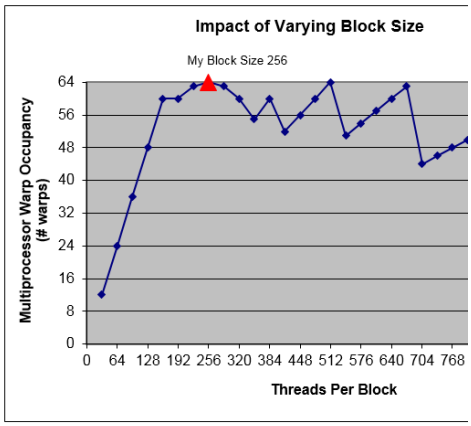
2.) Enter your resource usage:		
Threads Per Block	256	(Help)
Registers Per Thread	32	
Shared Memory Per Block (bytes)	4096	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	3.5
Threads per Warp	32
Warps per Multiprocessor	64
Threads per Multiprocessor	2048
Thread Blocks per Multiprocessor	16
Total # of 32-bit registers per Multiprocessor	65536
Register allocation unit size	256
Register allocation granularity	warp

Your chosen resource usage is indicated by the red triangle on the other data points represent the range of possible block sizes, regis shared memory allocation.



Impact of Varying Register Count Per Threa

CUPTI Library

The NVIDIA CUDA Profiling Tools Interface (CUPTI) provides performance analysis tools with detailed information about how applications are using the GPUs in a system. CUPTI provides two simple yet powerful mechanisms that allow performance analysis tools such as the NVIDIA Visual Profiler, TAU and Vampir Trace to understand the inner workings of an application and deliver valuable insights to developers.
The first mechanism is a callback API that allows tools to inject analysis code into the entry and exit point of each CUDA C Runtime (CUDART) and CUDA Driver API function. Using this callback API, tools can monitor an application's interactions with the CUDA Runtime and driver. The second mechanism allows performance analysis tools to query and configure hardware event counters designed into the GPU and software event counters in the CUDA driver. These event counters record activity such as instruction counts, memory transactions, cache hits/misses, divergent branches, and more.

Key Features

- Trace CUDA API usage by registering callbacks for API calls of interest
- Full support for entry and exit points in the CUDA C Runtime (CUDART) and CUDA Driver
- Sample hardware and software event counters, including:
- Instruction count and throughput
- Memory load/store events and throughput
- Cache hits/misses
- Branches and divergent branches
- Many more
- Enables automated bottleneck identification based on metrics such as instruction throughput, memory throughput, and more
- Normalized timestamps for CPU and GPU events

See the CUPTI User's Guide for a complete listing of hardware and software event counters available for performance analysis tools.

nvprof supports dumping the profile to a file which can be later imported into nvvp. To generate a profile for a MPI+CUDA application I simply start nvprof with the MPI launcher.

```
$ mpirun -np 2 nvprof -o simpleMPI.%q{OMPI_COMM_WORLD_RANK}.nvprof ./simpleMPI
Running on 2 nodes
==18811== NVPROF is profiling process 18811, command: ./simpleMPI
==18813== NVPROF is profiling process 18813, command: ./simpleMPI
Average of square roots is: 0.667279
PASSED
==18813== Generated result file: simpleMPI.1.nvprof
==18811== Generated result file: simpleMPI.0.nvprof
```

The output files produced by nvprof can be either read by nvprof to analyze the profile one rank at a time (using --import-profile) or imported into nvvp.

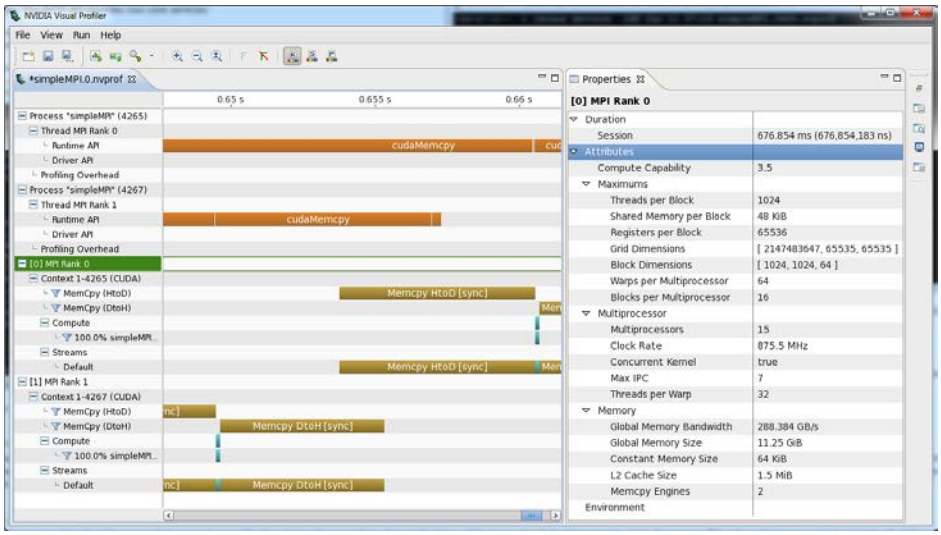


Figure – cupti profiling for CUDA + MPI binary

NVTX(NVIDIA Tools Extension) Library

The NVIDIA Tools Extension (NVTX) is an application interface to the NVIDIA Profiling tools, including the vNVIDIA Visual Profiler, NSight Eclipse Edition, and NSight Visual Studio Edition. NVTX allows you to annotate the profiler timeline with events and ranges and to customize their appearance and assign names to resources such as CPU threads and devices.

Use NVTX like any other C library: include the header "nvToolsExt.h", call the API functions from your source and link the NVTX library on the compiler command line with -lnvToolsExt. When we run this application in the NVIDIA Visual Profiler we get a timeline like the following image.

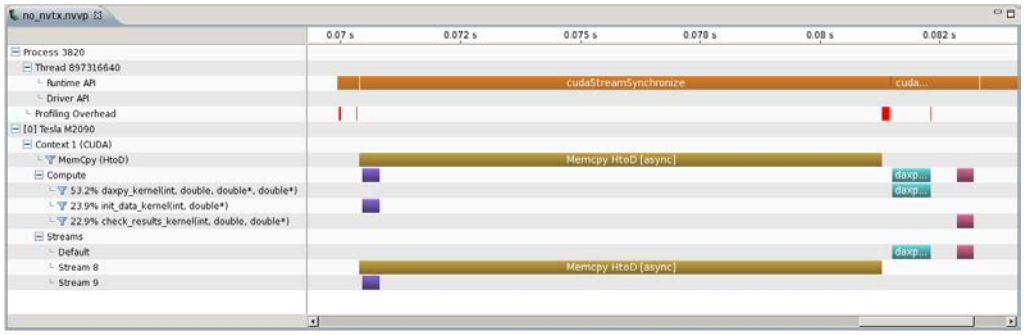


Figure – timeline view on nvvp for profiling output with NVTX library

To see the duration of init_host_data you can use nvtxRangePushA and nvtxRangePop:

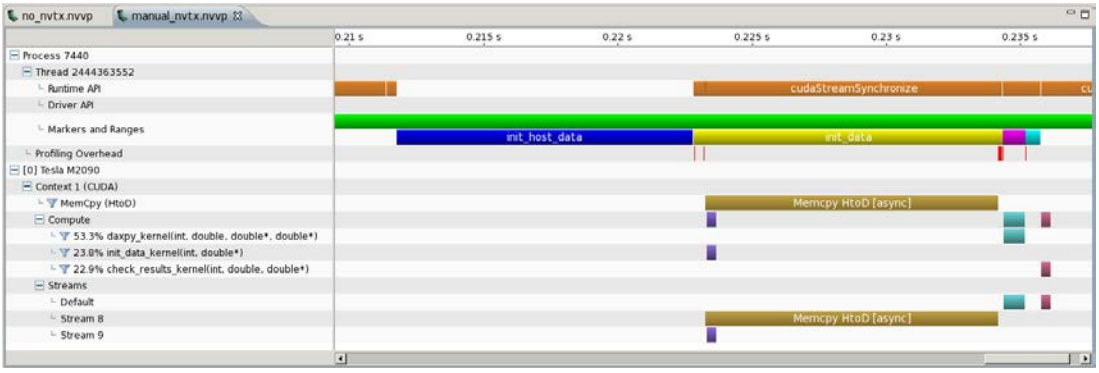
```
#include "nvToolsExt.h"
...
void init_host_data( int n, double * x ) {
   nvtxRangePushA("init_host_data");
   //initialize x on host
   ...
```

```
    nvtxRangePop();
}
...
```

All ranges created with nvtxRangePushA will be colored green. In an application which defines many ranges this default color can quickly become confusing, so NVTX offers the nvtxRangePushExfunction, which allows you to customize the color and appearance of a range in the timeline. For convenience in my own applications, I use the following macros to insert calls to nvtxRangePushExand nvtxRangePop.

```
#ifdef USE_NVTX
#include "nvToolsExt.h"

const uint32_t colors[] = { 0x0000ff00, 0x000000ff, 0x00ffff00, 0x00ff00ff, 0x0000ffff,
0x00ff0000, 0x00ffffff };
const int num_colors = sizeof(colors)/sizeof(uint32_t);

#define PUSH_RANGE(name,cid) { \
   int color_id = cid; \
   color_id = color_id%num_colors;\
   nvtxEventAttributes_t eventAttrib = {0}; \
   eventAttrib.version = NVTX_VERSION; \
   eventAttrib.size = NVTX_EVENT_ATTRIB_STRUCT_SIZE; \
   eventAttrib.colorType = NVTX_COLOR_ARGB; \
   eventAttrib.color = colors[color_id]; \
   eventAttrib.messageType = NVTX_MESSAGE_TYPE_ASCII; \
   eventAttrib.message.ascii = name; \
   nvtxRangePushEx(&eventAttrib); \
}
#define POP_RANGE nvtxRangePop();
#else
#define PUSH_RANGE(name,cid)
#define POP_RANGE
#endif
```

Using the macros for all host functions generates a timeline like the following image.



APPENDIX CODE EXAPLE

Example of PI serial C code

```
#include <stdio.h>
#include <math.h>

#define NN 1000000000


double integrate(int n) {
  double sum, h, x;
  int i;

  sum = 0.0;
  h   = 1.0 / (double) n;
  for (i = 1; i <= n; i++) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}

int main() {
  int n=NN;
  double PI25DT = 3.14159265358979323846264 3;
  double pi;
  pi = integrate(n);

  printf("pi is               %.16f\n", PI25DT);
  printf("pi is approximately %.16f\n", pi);
  printf("error is            %.16f with %d iteration\n", fabs(pi - PI25DT), n);

  return 0;
```

Example of Pi OpenMP

```
  #include <stdio.h>
  #include <math.h>
  #include <omp.h>

  #define NN 1000000000


  double integrate(int n) {
    double sum, h, x;
    int i;

    sum = 0.0;
    h   = 1.0 / (double) n;
  #pragma omp parallel for reduction(+:sum) private(x)
    for (i = 1; i <= n; i++) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
```

```
  }
  return sum * h;
}

int main() {
  int n=NN;
  double PI25DT = 3.14159265358979323846264 3;
  double pi;
  pi = integrate(n);

  printf("pi is               %.16f\n", PI25DT);
  printf("pi is approximately %.16f\n", pi);
  printf("error is            %.16f with %d iteration\n", fabs(pi - PI25DT), n);

  return 0;
}
```

Example of Pi OpenACC

```
#include <stdio.h>
#include <math.h>
#include "openacc.h"

#define NN 1000000000


double integrate(int n) {
  double sum, h, x;
  int i;

  sum = 0.0;
  h   = 1.0 / (double) n;
#pragma acc kernels
  for (i = 1; i <= n; i++) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
  }
  return sum * h;
}

int main() {
  int n=NN;
  double PI25DT = 3.14159265358979323846264 3;
                  double pi;
  pi = integrate(n);

  printf("pi is               %.16f\n", PI25DT);
  printf("pi is approximately %.16f\n", pi);
  printf("error is            %.16f with %d iteration\n", fabs(pi - PI25DT), n);

  return 0;
```

Example of Pi CUDA

```
  #include <stdio.h>
  #include <math.h>

  #define NN 1000000000

  void cudaErr(){
      printf("%s\n", cudaGetErrorString(cudaGetLastError()) );
      return ;
  }

  double integrate(int n) {
    double sum, h, x;
    int i;

    sum = 0.0;
    h   = 1.0 / (double) n;
    for (i = 1; i <= n; i++) {
      x = h * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    return sum * h;
  }


  __global__
  void  integrate_kernel(int n,  double * pi_gpu) {

   int idx = blockIdx.x * blockDim.x + threadIdx.x;
   int job_size = blockDim.x * gridDim.x;

    double sum, h, h_d, x;
    int i;
    int n_d = n / (job_size);

    sum = 0.0;
    pi_gpu[idx]=0.0;
    h   = 1.0 / (double) n;
    h_d = 1.0 / (double) n_d;
    for (i = idx +1; i <= n_d ; i += job_size ) {
      x = h_d * ((double)i - 0.5);
      sum += 4.0 / (1.0 + x*x);
    }
    pi_gpu[idx] =  sum * h_d;
  }



  double integrate_gpu(int n) {

    double *pi_cpu;
    double *pi_gpu;
    double pi =0.0;
    int bs = 100;
    int ts = 100;
    pi_cpu = (double *) malloc(  sizeof(double) * bs * ts );
    cudaMalloc( (void**)&pi_gpu, sizeof(double) * bs * ts );cudaErr();
    cudaMemset( pi_gpu, 0.0,     sizeof(double) * bs * ts );cudaErr();
    integrate_kernel <<< bs ,ts >>> ( n, pi_gpu);          cudaErr();
    cudaMemcpy( pi_cpu, pi_gpu, sizeof(double) * bs * ts , cudaMemcpyDeviceToHost);
cudaErr();
```

```
  cudaFree(pi_gpu); cudaErr();

  for( int i =0; i < bs * ts; i++) pi += pi_cpu[i];  //reduce

  free(pi_cpu);

  return pi ;
}


int main() {
  int n=NN;
  double PI25DT = 3.14159265358979323846264 3;

  double pi;
  pi = integrate_gpu(n);

  printf("pi is               %.16f\n", PI25DT);
  printf("pi is approximately %.16f\n", pi);
  printf("error is            %.16f with %d iteration\n", fabs(pi - PI25DT), n);

  return 0;
}
```

Laplace 2D

```
#include <math.h>
#include <string.h>
#include <openacc.h>
#include "timer.h"
#define NN 4096
#define NM 4096
double A[NN][NM];
double Anew[NN][NM];
int main(int argc, char** argv)
{
    const int n = NN;
    const int m = NM;
    const int iter_max = 1000;

    const double tol = 1.0e-6;
    double error     = 1.0;

    memset(A, 0, n * m * sizeof(double));
    memset(Anew, 0, n * m * sizeof(double));

    for (int j = 0; j < n; j++)    {
        A[j][0]    = 1.0;
        Anew[j][0] = 1.0;
    }

    printf("Jacobi relaxation Calculation: %d x %d mesh\n", n, m);

    StartTimer();
    int iter = 0;

    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

#pragma omp parallel for shared(m, n, Anew, A)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp parallel for shared(m, n, Anew, A)
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
```

```
        iter++;
    }
    double runtime = GetTimer();
    printf(" total: %f s\n", runtime / 1000);
}
```

Laplace 2D OpenACC

```
#include <math.h>
#include <string.h>
#include "timer.h"

#define NN 4096
#define NM 4096

double A[NN][NM];
double Anew[NN][NM];

int main(int argc, char** argv)
{
    const int n = NN;
    const int m = NM;
    const int iter_max = 1000;

    const double tol = 1.0e-6;
    double error     = 1.0;

    memset(A, 0, n * m * sizeof(double));
    memset(Anew, 0, n * m * sizeof(double));

    for (int j = 0; j < n; j++)
    {
        A[j][0]    = 1.0;
        Anew[j][0] = 1.0;
    }

    printf("Jacobi relaxation Calculation: %d x %d mesh\n", n, m);

    StartTimer();
    int iter = 0;

    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
        for( int j = 1; j < n-1; j++)
```

```
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);
        iter++;
    }

    double runtime = GetTimer();
    printf(" total: %f s\n", runtime / 1000);
}
```

Laplace 2D OpenACC Data Loop

```
#include <math.h>
#include <string.h>
#include "timer.h"

#define NN 4096
#define NM 4096

double A[NN][NM];
double Anew[NN][NM];

int main(int argc, char** argv)
{
    const int n = NN;
    const int m = NM;
    const int iter_max = 1000;

    const double tol = 1.0e-6;
    double error     = 1.0;

    memset(A, 0, n * m * sizeof(double));
    memset(Anew, 0, n * m * sizeof(double));

    for (int j = 0; j < n; j++)
    {
        A[j][0]    = 1.0;
        Anew[j][0] = 1.0;
    }

    printf("Jacobi relaxation Calculation: %d x %d mesh\n", n, m);

    StartTimer();
    int iter = 0;

#pragma acc data copy(A), create(Anew)
    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
        for( int j = 1; j < n-1; j++)
        {
```

```
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp parallel for shared(m, n, Anew, A)
#pragma acc kernels
        for( int j = 1; j < n-1; j++)
        {
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }
    double runtime = GetTimer();
    printf(" total: %f s\n", runtime / 1000);
}
```

Laplace 2D OpenACC Kernel Optimization

```
#include <math.h>
#include <string.h>
#include "timer.h"

#define n 4096
#define m 4096

double A[n][m];
double Anew[n][m];

int main(int argc, char** argv)
{
    const int iter_max = 1000;

    const double tol = 1.0e-6;
    double error     = 1.0;

    memset(A, 0, n * m * sizeof(double));
    memset(Anew, 0, n * m * sizeof(double));

    for (int j = 0; j < n; j++)
    {
        A[j][0]    = 1.0;
        Anew[j][0] = 1.0;
    }

    printf("Jacobi relaxation Calculation: %d x %d mesh\n", n, m);

    StartTimer();
```

```
    int iter = 0;

#pragma acc data copy(A), create(Anew)
    while ( error > tol && iter < iter_max )
    {
        error = 0.0;

#pragma omp parallel for shared(Anew, A)
#pragma acc kernels loop
        for( int j = 1; j < n-1; j++)
        {
#pragma acc loop gang(16) vector(32)
            for( int i = 1; i < m-1; i++ )
            {
                Anew[j][i] = 0.25 * ( A[j][i+1] + A[j][i-1]
                                    + A[j-1][i] + A[j+1][i]);
                error = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

#pragma omp parallel for shared(Anew, A)
#pragma acc kernels loop
        for( int j = 1; j < n-1; j++)
        {
#pragma acc loop gang(16) vector(32)
            for( int i = 1; i < m-1; i++ )
            {
                A[j][i] = Anew[j][i];
            }
        }

        if(iter % 100 == 0) printf("%5d, %0.6f\n", iter, error);

        iter++;
    }
    double runtime = GetTimer();
    printf(" total: %f s\n", runtime / 1000);
}
```

CUDA Library Examples

cuBLAS example

```
/* Includes, system */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* Includes, cuda */
#include <cuda_runtime.h>
#include <cublas_v2.h>

/* Matrix size */
#define M  (2048)
#define N  (1792)
#define K  (2048)


/* Main */
int main(int argc, char **argv)
{

        cublasStatus_t status;
        cublasHandle_t handle;
        float ms = 0.0;
        float s = 0.0;
        // for timer
        cudaEvent_t start, stop;

        float *h_A,*h_B,*h_, *h_C_ref;

        float *d_A ,*d_B,*d_C;

        float alpha = 1.0f;
        float beta = 0.0f;

        int nA = N * K;
        int nB = K * M;
        int nC = N * M;

        cudaEventCreate(&start);
        cudaEventCreate(&stop);

        /* Allocate host memory for the matrices */
        h_A = (float *)malloc(nA * sizeof(h_A[0]));
        h_B = (float *)malloc(nB * sizeof(h_B[0]));
        h_C = (float *)malloc(nC * sizeof(h_C[0]));

        /* Fill the matrices with test data */
        for (i = 0; i < nA; i++){    h_A[i] = rand() / (float)RAND_MAX;   }
        for (i = 0; i < nB; i++){    h_B[i] = rand() / (float)RAND_MAX;   }
        for (i = 0; i < nC; i++){    h_C[i] = rand() / (float)RAND_MAX;   }

        /* Initialize CUBLAS */
        status = cublasCreate(&handle);

        printf("GPU Malloc..\n");

        cudaMalloc((void **)&d_A, nA * sizeof(d_A[0]));
        cudaMalloc((void **)&d_B, nB * sizeof(d_B[0]));
        cudaMalloc((void **)&d_C, nC * sizeof(d_C[0]));
```

```
        /* Initialize the device matrices with the host matrices */
        cublasSetVector(nA, sizeof(h_A[0]), h_A, 1, d_A, 1);
        cublasSetVector(nB, sizeof(h_B[0]), h_B, 1, d_B, 1);
        cublasSetVector(nC, sizeof(h_C[0]), h_C, 1, d_C, 1);

        /* Performs operation using cublas */
        cudaEventRecord(start,0);
        cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, M, N, K, &alpha, d_A, M, d_B, K,
                    &beta, d_C, M);

        cudaEventRecord(stop, 0);
        cudaEventSynchronize(stop);
        cudaEventElapsedTime(&ms, start, stop);

        /* Read the result back */
        status = cublasGetVector(nC, sizeof(h_C[0]), d_C, 1, h_C, 1);

        /* Memory clean up */
        free(h_A);     free(h_B);     free(h_C);
        cudaFree(d_A); cudaFree(d_B); cudaFree(d_C);

        /* Shutdown */
        status = cublasDestroy(handle);

        return 0;
}
```

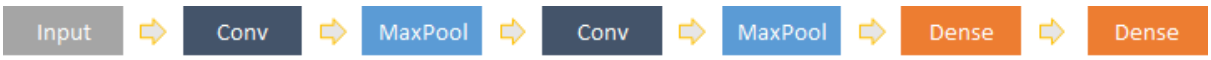
cuFFT example

```
void runTestcuFFT(int argc, char **argv)
{

    // Allocate host memory for the signal and filter
    Complex *h_signal = (Complex *)malloc(sizeof(Complex) * SIGNAL_SIZE);
    Complex *h_filter_kernel = (Complex *)malloc(sizeof(Complex) * FILTER_KERNEL_SIZE);

    // Initalize the memory for the signal
    for (unsigned int i = 0; i < SIGNAL_SIZE; ++i) {
        h_signal[i] .x = rand() / (float)RAND_MAX;
        h_signal[i].y = 0;
    }

    // Initalize the memory for the filter
    for (unsigned int i = 0; i < FILTER_KERNEL_SIZE; ++i) {
        h_filter_kernel[i].x = rand() / (float)RAND_MAX;
        h_filter_kernel[i].y = 0;
    }

    // Pad signal and filter kernel
    Complex *h_padded_signal;
    Complex *h_padded_filter_kernel;
    int new_size = PadData(h_signal, &h_padded_signal, SIGNAL_SIZE,
                           h_filter_kernel, &h_padded_filter_kernel,
FILTER_KERNEL_SIZE);
    int mem_size = sizeof(Complex) * new_size;

    // Allocate device memory for signal
    Complex *d_signal;
    cudaMalloc((void **)&d_signal, mem_size);

    // Copy host memory to device
    cudaMemcpy(d_signal, h_padded_signal, mem_size,
               cudaMemcpyHostToDevice);

    // Allocate device memory for filter kernel
    Complex *d_filter_kernel;
    cudaMalloc((void **)&d_filter_kernel, mem_size);

    // Create CUFFT plan
    cufftHandle plan;
    cufftPlan1d(&plan, new_size, CUFFT_C2C, 1);


    // Copy host memory to device
    cudaMemcpy(d_filter_kernel, h_padded_filter_kernel, mem_size,
               cudaMemcpyHostToDevice);

    // Transform signal and kernel
    printf("Transforming signal cufftExecC2C\n");
    cufftExecC2C(plan, (cufftComplex *)d_signal, (cufftComplex *)d_signal,
CUFFT_FORWARD);
    cufftExecC2C(plan, (cufftComplex *)d_filter_kernel, (cufftComplex *)d_filter_kernel,
                 CUFFT_FORWARD);


    // Multiply the coefficients together and normalize the result
    printf("Performing point-wise complex multiply and scale.\n");
    cudaEventRecord(startPointwise, 0);
```

```
    complexPointwiseMulAndScale(new_size, (float *restrict)d_signal,
                                (float *restrict)d_filter_kernel);

    // Transform signal back
    printf("Transforming signal back cufftExecC2C\n");
    cufftExecC2C(plan, (cufftComplex *)d_signal, (cufftComplex *)d_signal,
CUFFT_INVERSE);


    // Copy device memory to host
    Complex *h_convolved_signal = h_padded_signal;
    cudaMemcpy(h_convolved_signal, d_signal, mem_size,
               cudaMemcpyDeviceToHost);


    // Allocate host memory for the convolution result
    Complex *h_convolved_signal_ref = (Complex *)malloc(sizeof(Complex) * SIGNAL_SIZE);

    // Convolve on the host
    printf("Performing Convolution on the host and checking correctness\n");
    Convolve(h_signal, SIGNAL_SIZE,
             h_filter_kernel, FILTER_KERNEL_SIZE,
             h_convolved_signal_ref);

    // check result
    int testResult = compareL2((float *)h_convolved_signal_ref,
                               (float *)h_convolved_signal,
                               2 * SIGNAL_SIZE, 1e-5f);


    //Destroy CUFFT context
    error = cufftDestroy(plan);
    if (error != CUFFT_SUCCESS)
        printf("CUFFT Error: %d", error);

    // cleanup memory
    free(h_signal);
    free(h_filter_kernel);
    free(h_padded_signal);
    free(h_padded_filter_kernel);
    free(h_convolved_signal_ref);
    cudaFree(d_signal);
    cudaFree(d_filter_kernel);
    cudaEventDestroy(start);
    cudaEventDestroy(stop);
    //cutilDeviceReset();
}
```

Example of cuDNN (심화과정)

```
cudnn.SpatialConvolution(nInputPlane, nOutputPlane, kW, kH, [dW = 1],
                         [dH = 1], [padW = 0], [padH = 0], [groups = 1])
cudnn.VolumetricConvolution(nInputPlane, nOutputPlane, kT, kW, kH, dT,
                            dW, dH, padT, padW, padH)
cudnn.SpatialMaxPooling(kW, kH, dW, dH, padW, padH)
cudnn.SpatialAveragePooling(kW, kH, dW, dH)
-- Functions
cudnn.ReLU(inplace[=false])
cudnn.Tanh(inplace[=false])
cudnn.Sigmoid(inplace[=false])
cudnn.SoftMax(fastMode [= false])
cudnn.SpatialSoftMax(fastMode [= false])
```




```
    convoluteForward(conv1, n, c, h, w, srcData, &dstData);
    poolForward(n, c, h, w, dstData, &srcData);

    convoluteForward(conv2, n, c, h, w, srcData, &dstData);
    poolForward(n, c, h, w, dstData, &srcData);

    fullyConnectedForward(ip1, n, c, h, w, srcData, &dstData);
    activationForward(n, c, h, w, dstData, &srcData);
    LeRuForward(n, c, h, w, srcData, &dstData);
    fullyConnectedForward(ip2, n, c, h, w, dstData, &srcData);
    softmaxForward(n, c, h, w, srcData, &dstData);
```

```
void convoluteForward(const Layer_t<value_type>& conv, int& n, int& c, int& h, int& w,
       value_type* srcData, value_type** dstData){

    setTensorDesc(srcTensorDesc, tensorFormat, dataType, n, c, h, w);

    const int tensorDims = 4;
    int tensorOuputDimA[tensorDims] = { n, c, h, w };
    const int filterDimA[tensorDims] = { conv.outputs, conv.inputs,
                                         conv.kernel_dim, conv.kernel_dim };

    cudnnSetFilterNdDescriptor(filterDesc, dataType, tensorDims, filterDimA);
    const int convDims = 2;
    int padA[convDims] = { 0, 0 };
    int filterStrideA[convDims] = { 1, 1 };
    int upscaleA[convDims] = { 1, 1 };
    cudnnSetConvolutionNdDescriptor(convDesc, convDims, padA, filterStrideA, upscaleA,
        CUDNN_CROSS_CORRELATION);

    cudnnGetConvolutionNdForwardOutputDim(convDesc, srcTensorDesc, filterDesc,
        tensorDims, tensorOuputDimA);
    n = tensorOuputDimA[0]; c = tensorOuputDimA[1];
    h = tensorOuputDimA[2]; w = tensorOuputDimA[3];

    setTensorDesc(dstTensorDesc, tensorFormat, dataType, n, c, h, w);
```

```
    cudnnGetConvolutionForwardAlgorithm(cudnnHandle, srcTensorDesc,
        filterDesc, convDesc, dstTensorDesc,
        CUDNN_CONVOLUTION_FWD_PREFER_FASTEST, 0, &algo );

return;
}
```

```
void poolForward(int& n, int& c, int& h, int& w,
        value_type* srcData, value_type** dstData){
    const int poolDims = 2;
    int windowDimA[poolDims] = { 2, 2 };
    int paddingA[poolDims] = { 0, 0 };
    int strideA[poolDims] = { 2, 2 };
    cudnnSetPoolingNdDescriptor(poolingDesc, CUDNN_POOLING_MAX,
       poolDims, windowDimA, paddingA, strideA);

    setTensorDesc(srcTensorDesc, tensorFormat, dataType, n, c, h, w);

    const int tensorDims = 4;
    int tensorOuputDimA[tensorDims] = { n, c, h, w };
    cudnnGetPoolingNdForwardOutputDim(poolingDesc, srcTensorDesc,
       tensorDims,tensorOuputDimA));

    n = tensorOuputDimA[0]; c = tensorOuputDimA[1];
    h = tensorOuputDimA[2]; w = tensorOuputDimA[3];

   setTensorDesc(dstTensorDesc, tensorFormat, dataType, n, c, h, w);

    resize(n*c*h*w, dstData);
    scaling_type alpha = scaling_type(1);
    scaling_type beta = scaling_type(0);

    cudnnPoolingForward(cudnnHandle, poolingDesc,
       &alpha,srcTensorDesc,srcData,
       &beta, dstTensorDesc, *dstData);
    return ;
}
```

```
void LeRuForward(int n, int c, int h, int w,
                  value_type* srcData, value_type** dstData)    {
    unsigned lrnN = 5;
    double lrnAlpha, lrnBeta, lrnK;
    lrnAlpha = 0.0001; lrnBeta = 0.75; lrnK = 1.0;
    cudnnSetLRNDescriptor(normDesc,lrnN, lrnAlpha, lrnBeta, lrnK);
    resize(n*c*h*w, dstData);

    setTensorDesc(srcTensorDesc, tensorFormat, dataType, n, c, h, w);
    setTensorDesc(dstTensorDesc, tensorFormat, dataType, n, c, h, w);

    scaling_type alpha = scaling_type(1);
    scaling_type beta = scaling_type(0);
    cudnnLRNCrossChannelForward(cudnnHandle,normDesc,CUDNN_LRN_CROSS_CHANNEL_DIM1,
        &alpha,srcTensorDesc,srcData,
        &beta,dstTensorDesc, *dstData);
    return ;
       }
```

```
void SoftmaxForward(int n, int c, int h, int w,
                    value_type* srcData, value_type** dstData){
    resize(n*c*h*w, dstData);

    setTensorDesc(srcTensorDesc, tensorFormat, dataType, n, c, h, w);
    setTensorDesc(dstTensorDesc, tensorFormat, dataType, n, c, h, w);

    scaling_type alpha = scaling_type(1);
    scaling_type beta = scaling_type(0);
    cudnnSoftmaxForward(cudnnHandle,
       CUDNN_SOFTMAX_ACCURATE, CUDNN_SOFTMAX_MODE_CHANNEL,
       &alpha, srcTensorDesc, srcData,
       &beta, dstTensorDesc, *dstData);
    return ;
}
```