A short introduction to GPU programming

Kees Lemmens,
Email: C.W.J.Lemmens@tudelft.nl,
Delft Institute for Applied Mathematics (DIAM),
Faculty of EEMCS, Delft University of Technology,
The Netherlands.

Part 1: Introduction to GPUs

Historical overview of parallel programming

1985-95: Super (vector) computers era (Convex, Cray, TM)

1995- : Beowulf concept: Linux based low budget PC's

2000- : Linux clusters: Linux based (very) high budget PC's

2004- : Vector computing on GPU cards designed for gaming

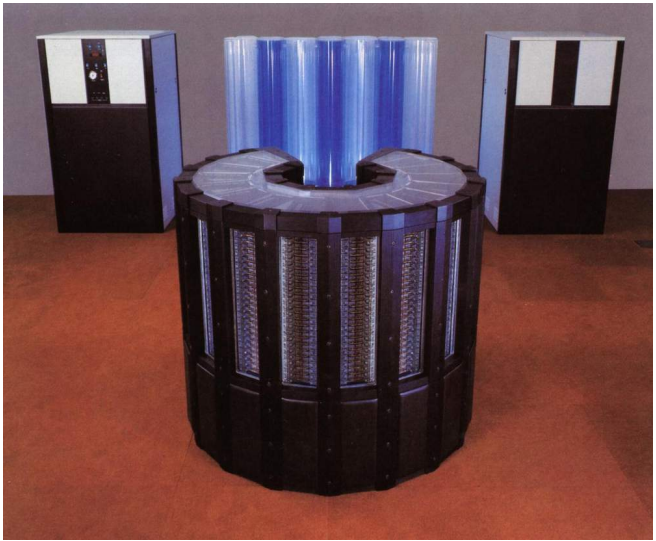
2008- : High end vector computing on dedicated GPU cards (Nvidia)

2010- : Multi GPU systems:

- Multicore systems with 4-8 GPUs
- Large GPU clusters with thousands of GPUs

2015- : Focus moving from Numerical Methods to Machine Learning and AI

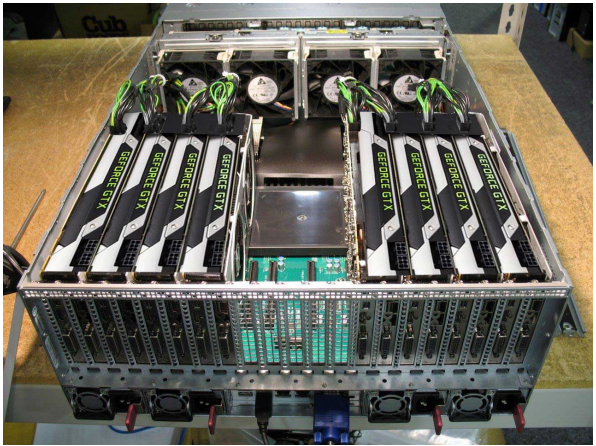
USA: NASA Cray-2 vector Computer, 1.9 Gigaflops, 250 kW (1985)



China: Tianhe-2, 34 Petaflops, 24 MWatt, 390 M$, 38.000 Xeon + 44.000 Xeon Phi (2016)

USA: Summit, 150 Petaflops, 13 MWatt, 325 M$, 9200 Power9 + 28.000 Tesla V100 GPU (2019)

USA: Maxwell 8 GPU desktop 20-30 Teraflops (SP), 1.6 kWatt, 15 k$, 8 GPUs GTX980 (2015)



USA: Bizon 4 GPU desktop, 65 Teraflops (SP), 1 kWatt, 3 k$, 4 GPUs RTX8000 (2022)

USA San Jose: Jetson Tegra X2 development kit,
30 Watt, 300 $, 1 GPU Pascal on Tegra with 4 core ARM 64 (2017)

Parallel computing on Graphical Processing Units 1

GPU: Use graphics (video) cards for scientific computing

Features:

- Based upon standard video cards by Nvidia (or ATI)
- Uses standard computer with either Linux, MSW or MacOS
- Programming model SIMD (Single Instruction, Multiple Data)
- Parallelisation inside card through "threads"
- Dedicated software to access card and start "kernels" (Cuda, OpenCL)

Most popular software toolkits are **CUDA** by Nvidia and **OpenCL** by Khronos, but we'll mainly talk about **CUDA**, as it is the most commonly used toolkit today.

Parallel computing on Graphics Cards 2

Advantages

- Hardware is cheap compared with Linux clusters (at least if gaming cards are used)
- Lower power consumption per Flop than CPU
- Simple GPU already available in many systems without any extra investment
- Capable of thousands to millions of parallel threads on a single GPU card
- Very fast for algorithms that can be efficiently parallelised (not that many ;-))
- Often better scalability than MPI for large problems due to faster communication
- New libraries hiding complexity to the user: **BLAS, FFTW, SPARSE, SOLVER**
- Easy integration with 3D graphics if code runs on the same card (or by using Prime)

Parallel computing on Graphics Cards 3

Disadvantages

- Limited amount of memory available (between 4-48 GByte)
- Memory transfers between host and GPU cause a performance bottleneck
- Often no ECC: error correcting memory (except expensive highend GPUs)
- Fast double precision GPUs are still quite expensive (P100, V100, A100)
- Slow for algorithms without enough data parallellism (e.g. scalar vector product)
- Debugging code on GPU can be complicated
- Achieving theoretically possible speedup is almost impossible
- Using Multi GPUs in a cluster is quite complex (often done with MPI or OpenMP)

What is Cuda?

Cuda: means "Compute Unified Device Architecture" and is a software toolkit by Nvidia to ease the use of (Nvidia) graphics cards for scientific programming. It needs on a special video driver as well.

What is Cuda?

Cuda: means "Compute Unified Device Architecture" and is a software toolkit by Nvidia to ease the use of (Nvidia) graphics cards for scientific programming. It needs on a special video driver as well.

Features:

- Special C compiler to build code both for CPU and GPU (nvcc)
- C Language extensions (codewords) :
  - distinguish CPU and GPU functions
  - access different types of memory on the GPU
  - specify how code should be parallelized on the GPU
- Library routines for memory transfer between host and GPU
- Extra BLAS, Sparse and FFT libraries for easy porting existing code
- Dedicated to Nvidia hardware
- Mainly standard C on the GPU (limited support for C++ and Fortran)

What is OpenCL?

OpenCL: means "Open Computing Language" and is a C99 like language for parallel kernels in combination with an API to get these kernels loaded and started on different types of parallel hardware (e.g. AMD, Intel, Nvidia).

What is OpenCL?

OpenCL: means "Open Computing Language" and is a C99 like language for parallel kernels in combination with an API to get these kernels loaded and started on different types of parallel hardware (e.g. AMD, Intel, Nvidia).

Features:

- Only contains a dedicated compiler for building GPU code
- So, no support for mixed (GPU + CPU) source code as in Cuda
- This makes starting kernels is more complicated than Cuda
- Data transfer between host and GPU is done using library functions
- Runs on several types of GPU (AMD, Nvidia) and is easily portable
- However, performance is **not necessarily portable** across platforms
- OpenCL is an Open Standard, while Cuda causes a vendor-lock to Nvidia
- Less mature than Cuda, often no or limited support for special GPU features

When to use GPUs? Matrix Product comparison

Let us look at a computation intensive Maxtrix product A x B in single precision.
Note that the time is in seconds.

Type	512	1024	2048	4096	8192	16384
Blas/Atlas Xeon 3.6	0.013	0.10	0.78	6.11	48.67	388
CuBlas GPU K620	0.001	0.005	0.0348	0.250	1.843	-
CuBlas GPU GTX480	0.0016	0.007	0.036	0.222	1.508	-
CuBlas GPU Kepler K20	0.0008	0.0034	0.0155	0.085	0.542	3.94
CuBlas GPU TitanX	0.0007	0.0028	0.0118	0.058	0.327	1.96
CuBlas GPU P100	0.0004	0.0014	0.0064	0.032	0.192	1.14
Ratio Xeon/P100	32x	71x	122x	190x	253x	340x

When to use GPUs? Matrix Product comparison

Let us look at a computation intensive Maxtrix product A x B in single precision.
Note that the time is in seconds.

Type	512	1024	2048	4096	8192	16384
Blas/Atlas Xeon 3.6	0.013	0.10	0.78	6.11	48.67	388
CuBlas GPU K620	0.001	0.005	0.0348	0.250	1.843	-
CuBlas GPU GTX480	0.0016	0.007	0.036	0.222	1.508	-
CuBlas GPU Kepler K20	0.0008	0.0034	0.0155	0.085	0.542	3.94
CuBlas GPU TitanX	0.0007	0.0028	0.0118	0.058	0.327	1.96
CuBlas GPU P100	0.0004	0.0014	0.0064	0.032	0.192	1.14
Ratio Xeon/P100	32x	71x	122x	190x	253x	340x

Conclusions:

- GPUs are much faster than CPUs, especially for larger dimensions.
- GPUs are very limited by their main memory compared to CPUs.
- Gaming cards (GTX480) good price/performance ratio, less memory.

Note : GTX480 and K20 cards in identical hostsystems perform the same (for single prec.)

When to use GPUs? Matrix Vector Product comparison

Now look at a less computation intensive Matrix-Vector product A x b in single precision.
Note that the time is in **milli**seconds.

Type (nc=no communication)	512	1024	2048	4096	16384	32768
Blas/Atlas Xeon 3.6	0.1	0.4	1.6	7	100	430
CuBlas GPU K620	0.3	0.9	4	14	213	-
CuBlas GPU GTX480	0.4	2	6	30	400	-
CuBlas GPU K20	0.7	1	3	10	200	800
CuBlas GPU TitanX	0.3	0.9	3	10	200	700
CuBlas GPU P100	0.17	0.44	1.5	5.6	89	350
CuBlas GPU P100 (nc)	0.05	0.05	0.09	0.17	2	7.9
Ratio Xeon/P100	0.6x	0.9x	1.1x	1.25x	1.12x	1.23x
Ratio Xeon/P100 (nc)	2x	8x	18x	41x	50x	54x

When to use GPUs? Matrix Vector Product comparison

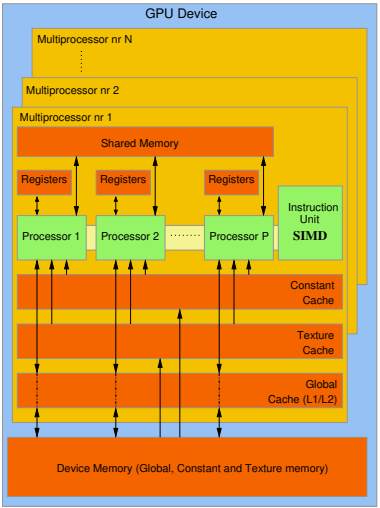
Now look at a less computation intensive Matrix-Vector product A x b in single precision.
Note that the time is in **milli**seconds.

Type (nc=no communication)	512	1024	2048	4096	16384	32768
Blas/Atlas Xeon 3.6	0.1	0.4	1.6	7	100	430
CuBlas GPU K620	0.3	0.9	4	14	213	-
CuBlas GPU GTX480	0.4	2	6	30	400	-
CuBlas GPU K20	0.7	1	3	10	200	800
CuBlas GPU TitanX	0.3	0.9	3	10	200	700
CuBlas GPU P100	0.17	0.44	1.5	5.6	89	350
CuBlas GPU P100 (nc)	0.05	0.05	0.09	0.17	2	7.9
Ratio Xeon/P100	0.6x	0.9x	1.1x	1.25x	1.12x	1.23x
Ratio Xeon/P100 (nc)	2x	8x	18x	41x	50x	54x

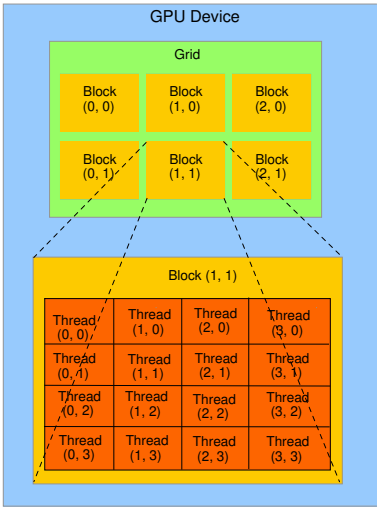
Conclusions :

- GPU performance doesn't improve much for larger dimensions.
- Difference in performance for GPUs is only small.
- Blas on CPU is faster than on GPU unless no host transfers are required (nc).

Hardware layout of the GPU



Processing layout of a GPU kernel



Comparsion to a Pasta Machine (9 threads per block ;-))

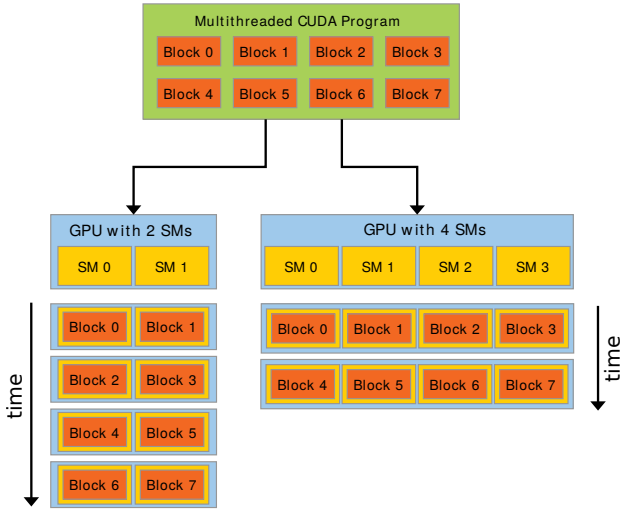
Relation software $\Longleftrightarrow$ hardware

Thread $\Longleftrightarrow$ **Core**

Block $\Longleftrightarrow$ **Multiprocessor**

Grid $\Longleftrightarrow$ **Device**

Scalability of a GPU kernel



Some capabilities of an Nvidia GPU

- An Nvidia GPU card can have 1 or more devices (Tesla S1070 = 4, Tesla C2070 = 1, NVS290 = 1, K20 = 1, K80 = 4, Titan X = 1)

Some capabilities of an Nvidia GPU

- An Nvidia GPU card can have 1 or more devices (Tesla S1070 = 4, Tesla C2070 = 1, NVS290 = 1, K20 = 1, K80 = 4, Titan X = 1)
- A single device can have several streamprocessors (SM) (K620 = 3, K20 = 13, K80 = 2x13, Titan X = 24, P100 = 28, V100=40)

Some capabilities of an Nvidia GPU

- An Nvidia GPU card can have 1 or more devices (Tesla S1070 = 4, Tesla C2070 = 1, NVS290 = 1, K20 = 1, K80 = 4, Titan X = 1)
- A single device can have several streamprocessors (SM) (K620 = 3, K20 = 13, K80 = 2x13, Titan X = 24, P100 = 28, V100=40)
- Each SM has 32 (2.0), 48 (2.x), 192 (3.x) or 128 (5-8.x) cores. A SM can run 32 active (semi) concurrent threads called warps

Some capabilities of an Nvidia GPU

- An Nvidia GPU card can have 1 or more devices (Tesla S1070 = 4, Tesla C2070 = 1, NVS290 = 1, K20 = 1, K80 = 4, Titan X = 1)
- A single device can have several streamprocessors (SM) (K620 = 3, K20 = 13, K80 = 2x13, Titan X = 24, P100 = 28, V100=40)
- Each SM has 32 (2.0), 48 (2.x), 192 (3.x) or 128 (5-8.x) cores. A SM can run 32 active (semi) concurrent threads called warps
- Threads in a SM run concurrently (max. 8-32) or sequentially if more. Maximum resident threads per SM is 1536 (2.x and up).

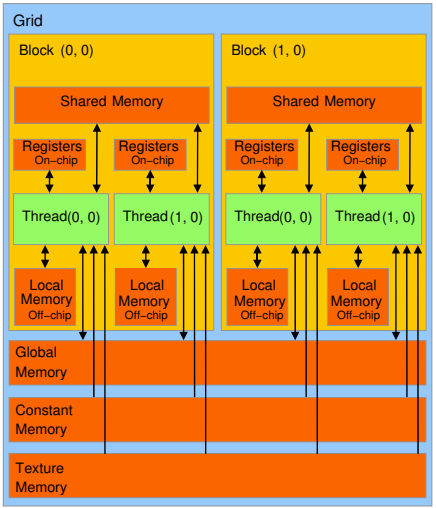
Some capabilities of an Nvidia GPU

- An Nvidia GPU card can have 1 or more devices (Tesla S1070 = 4, Tesla C2070 = 1, NVS290 = 1, K20 = 1, K80 = 4, Titan X = 1)
- A single device can have several streamprocessors (SM) (K620 = 3, K20 = 13, K80 = 2x13, Titan X = 24, P100 = 28, V100=40)
- Each SM has 32 (2.0), 48 (2.x), 192 (3.x) or 128 (5-8.x) cores. A SM can run 32 active (semi) concurrent threads called warps
- Threads in a SM run concurrently (max. 8-32) or sequentially if more. Maximum resident threads per SM is 1536 (2.x and up).
- A block is a set of threads running on a single SM. Maximum number of threads per block is 512-1024 (x and y) and 64 (z). The total maximum (x*y*z) is 1024 (2.x and up)

Some capabilities of an Nvidia GPU

- An Nvidia GPU card can have 1 or more devices (Tesla S1070 = 4, Tesla C2070 = 1, NVS290 = 1, K20 = 1, K80 = 4, Titan X = 1)
- A single device can have several streamprocessors (SM) (K620 = 3, K20 = 13, K80 = 2x13, Titan X = 24, P100 = 28, V100=40)
- Each SM has 32 (2.0), 48 (2.x), 192 (3.x) or 128 (5-8.x) cores. A SM can run 32 active (semi) concurrent threads called warps
- Threads in a SM run concurrently (max. 8-32) or sequentially if more. Maximum resident threads per SM is 1536 (2.x and up).
- A block is a set of threads running on a single SM. Maximum number of threads per block is 512-1024 (x and y) and 64 (z). The total maximum (x*y*z) is 1024 (2.x and up)
- Maximum number of blocks is $2^{31} - 1$ for each dimension (for arch 2.x $2^{16} - 1$) independent of SM count. Arch 2.x and up have 3 dimensions (so x, y and z)

Memory organisation of a GPU 1



Memory organisation of a GPU 2

- Device memory consists of global, constant and texture memory

Memory organisation of a GPU 2

- Device memory consists of global, constant and texture memory
- Global memory accessible by all streaming multiprocessors and is persistent

Memory organisation of a GPU 2

- Device memory consists of global, constant and texture memory
- Global memory accessible by all streaming multiprocessors and is persistent
- Constant and texture memory can only be *read* and have a **local cache** on each SM

Memory organisation of a GPU 2

- Device memory consists of global, constant and texture memory
- Global memory accessible by all streaming multiprocessors and is persistent
- Constant and texture memory can only be *read* and have a **local cache** on each SM
- Each SM has shared memory (on-chip) that can be used by all threads in an active block but is non-persistent (kind of cache)

Memory organisation of a GPU 2

- Device memory consists of global, constant and texture memory
- Global memory accessible by all streaming multiprocessors and is persistent
- Constant and texture memory can only be *read* and have a **local cache** on each SM
- Each SM has shared memory (on-chip) that can be used by all threads in an active block but is non-persistent (kind of cache)
- Each SM has local memory (off-chip) and registers (on-chip) that can only be used by a single thread and which is also non-persistent

Memory organisation of a GPU 2

- Device memory consists of global, constant and texture memory
- Global memory accessible by all streaming multiprocessors and is persistent
- Constant and texture memory can only be *read* and have a **local cache** on each SM
- Each SM has shared memory (on-chip) that can be used by all threads in an active block but is non-persistent (kind of cache)
- Each SM has local memory (off-chip) and registers (on-chip) that can only be used by a single thread and which is also non-persistent
- Note that memory access times also depend on the access method! Coalesced access may give a large performance boost, especially for less computation intensive problems
- Coalesced access is likely much better if the number of threads per block is always a multiple of the warp size (32 threads)

New features for GPUs architectures

- 1.x or Tesla: Now obsolete, name still used as a brandname.
- 2.x or Fermi: 48 cores per SM, atomic operations. L1 cache (combined with Shared Memory) on each SM and a single L2 cache per device for global memory. Explicit use of shared memory less important.

New features for GPUs architectures

- 1.x or Tesla: Now obsolete, name still used as a brandname.
- 2.x or Fermi: 48 cores per SM, atomic operations. L1 cache (combined with Shared Memory) on each SM and a single L2 cache per device for global memory. Explicit use of shared memory less important.
- 3.x or Kepler: 192 cores per SM, L1 cache only for local memory, Only L2 cache for global memory. Unified Memory Access (no explicit host- device transfers). Explicit use of shared memory again more important.

New features for GPUs architectures

- 1.x or Tesla: Now obsolete, name still used as a brandname.
- 2.x or Fermi: 48 cores per SM, atomic operations. L1 cache (combined with Shared Memory) on each SM and a single L2 cache per device for global memory. Explicit use of shared memory less important.
- 3.x or Kepler: 192 cores per SM, L1 cache only for local memory, Only L2 cache for global memory. Unified Memory Access (no explicit host- device transfers). Explicit use of shared memory again more important.
- 5.x or Maxwell: 128 cores per SM, L1 cache independent from Shared Memory. Dynamic Parallellism: start kernels from a running kernel.

New features for GPUs architectures

- 1.x or Tesla: Now obsolete, name still used as a brandname.
- 2.x or Fermi: 48 cores per SM, atomic operations. L1 cache (combined with Shared Memory) on each SM and a single L2 cache per device for global memory. Explicit use of shared memory less important.
- 3.x or Kepler: 192 cores per SM, L1 cache only for local memory, Only L2 cache for global memory. Unified Memory Access (no explicit host- device transfers). Explicit use of shared memory again more important.
- 5.x or Maxwell: 128 cores per SM, L1 cache independent from Shared Memory. Dynamic Parallellism: start kernels from a running kernel.
- 6.x or Pascal: 128 cores per SM, Half precision FP (16-bit), NVLink bus to directly link GPUs, bypassing PCI-E bus (5-15 times faster).

New features for GPUs architectures

- 1.x or Tesla: Now obsolete, name still used as a brandname.
- 2.x or Fermi: 48 cores per SM, atomic operations. L1 cache (combined with Shared Memory) on each SM and a single L2 cache per device for global memory. Explicit use of shared memory less important.
- 3.x or Kepler: 192 cores per SM, L1 cache only for local memory, Only L2 cache for global memory. Unified Memory Access (no explicit host- device transfers). Explicit use of shared memory again more important.
- 5.x or Maxwell: 128 cores per SM, L1 cache independent from Shared Memory. Dynamic Parallellism: start kernels from a running kernel.
- 6.x or Pascal: 128 cores per SM, Half precision FP (16-bit), NVLink bus to directly link GPUs, bypassing PCI-E bus (5-15 times faster).
- 7.0 or Volta: more half precision FP, Tensor cores (deep learning).
- 7.5 or Turing: mainly graphic and raytracing improvements.
- 8.0 or Ampere: several Tensor core improvements.

Example of parallel work: Amish building a barn

Example of parallel work: Amish raising a barn 2

Example of parallel work

If a single Amish can build a barn in 50 days then
50 Amish can build that barn in a single day!

Example of parallel work

If a single Amish can build a barn in 50 days then
50 Amish can build that barn in a single day!

However, in case of ordinary people **and** for GPUs
this is far to optimistic ...

Parallel matrix summation analysis

Suppose we need the sum of 2 (large) matrices A and B :

$$\begin{pmatrix} A(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & A(N,N) \end{pmatrix} + \begin{pmatrix} B(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & B(N,N) \end{pmatrix} = C$$

How could we parallelize this and what would be the speedup?

Parallel matrix summation: speedup using strips

Sequential calculation: $N.N$ elements, 1 operation/element:

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel matrix summation: speedup using strips

Sequential calculation: $N.N$ elements, 1 operation/element:

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips: $N.\frac{N}{P}$ elements, 1 op/el:

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel matrix summation: speedup using strips

Sequential calculation: $N.N$ elements, 1 operation/element:

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips: $N.\frac{N}{P}$ elements, 1 op/el:

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel communication for strips: scatter A + B + gather C :

$$T_{com} = P.3.\frac{N^2}{P}.t_2 = O(N^2)$$

Parallel matrix summation: speedup using strips

Sequential calculation: $N.N$ elements, 1 operation/element:

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips: $N.\frac{N}{P}$ elements, 1 op/el:

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel communication for strips: scatter $A + B$ + gather C :

$$T_{com} = P.3.\frac{N^2}{P}.t_2 = O(N^2)$$

$$\text{For } (P > \frac{t_1}{t_2}) : \ T_{par} = T_{cal} + T_{com} = O(\frac{N^2}{P}) + O(N^2) \approx O(N^2)$$

Parallel matrix summation: speedup using strips

Sequential calculation: $N.N$ elements, 1 operation/element:

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips: $N.\frac{N}{P}$ elements, 1 op/el:

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel communication for strips: scatter A + B + gather C :

$$T_{com} = P.3.\frac{N^2}{P}.t_2 = O(N^2)$$

$$\text{For } (P > \frac{t_1}{t_2}) : \; T_{par} = T_{cal} + T_{com} = O(\frac{N^2}{P}) + O(N^2) \approx O(N^2)$$

$$\text{Speedup} : \; \frac{T_{seq}}{T_{par}} = \frac{O(N^2)}{O(N^2)} = O(1), \text{ some for } P < \frac{t_1}{3.t_2} \text{ (no } N \text{ dep. !)}$$

Parallel matrix vector product analysis

Suppose we need the product of a (large) matrix A and vector x :

$$\begin{pmatrix} A(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & A(N,N) \end{pmatrix} * \begin{pmatrix} x(1) \\ \dots \\ x(N) \end{pmatrix} = b$$

How could we parallelize this and what would be the speedup?

Parallel matrix vector product analysis: speedup using strips

Sequential calculation: N computations for each element of x of size N :

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel matrix vector product analysis: speedup using strips

Sequential calculation: N computations for each element of x of size N :

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips: N computations for $\frac{N}{P}$ elements of x :

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel matrix vector product analysis: speedup using strips

Sequential calculation: N computations for each element of x of size N :

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips: N computations for $\frac{N}{P}$ elements of x :

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel communication for strips: scatter $A + x +$ gather b :

$$T_{com} = P.(\frac{N^2}{P} + 2.\frac{N}{P}).t_2 = O(N^2)$$

Parallel matrix vector product analysis: speedup using strips

Sequential calculation: N computations for each element of x of size N :

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips: N computations for $\frac{N}{P}$ elements of x :

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel communication for strips: scatter $A + x$ + gather b :

$$T_{com} = P.(\frac{N^2}{P} + 2.\frac{N}{P}).t_2 = O(N^2)$$

For $(P >> \frac{t_1}{t_2})$: $T_{par} = T_{cal} + T_{com} = O(\frac{N^2}{P}) + O(N^2) \approx O(N^2)$

Parallel matrix vector product analysis: speedup using strips

Sequential calculation: N computations for each element of x of size N :

$$T_{seq} = N^2.t_1 = O(N^2)$$

Parallel calculation for strips: N computations for $\frac{N}{P}$ elements of x :

$$T_{cal} = \frac{N^2}{P}.t_1 = O(\frac{N^2}{P})$$

Parallel communication for strips: scatter $A + x$ + gather b :

$$T_{com} = P.(\frac{N^2}{P} + 2.\frac{N}{P}).t_2 = O(N^2)$$

$$\text{For } (P >> \frac{t_1}{t_2}) : \ T_{par} = T_{cal} + T_{com} = O(\frac{N^2}{P}) + O(N^2) \approx O(N^2)$$

$$\text{Speedup : } \frac{T_{seq}}{T_{par}} = \frac{O(N^2)}{O(N^2)} = O(1), \text{ some speedup for } P < \frac{t_1}{t_2} \text{ (no } N \text{ dep. !)}$$

Parallel matrix multiplication analysis

Suppose we need the product of 2 (large) matrices A and B:

$$\begin{pmatrix} A(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & A(N,N) \end{pmatrix} * \begin{pmatrix} B(1,1) & \dots & \dots \\ \dots & \dots & \dots \\ \dots & \dots & B(N,N) \end{pmatrix} = C$$

How could we parallelize this and what would be the speedup?

Parallel matrix multiplication: speedup using strips

Sequential calculation : $N.N$ elements, N operations/element:

$$T_{seq} = N^3.t_1 = O(N^3)$$

Parallel matrix multiplication: speedup using strips

Sequential calculation : $N.N$ elements, N operations/element:

$$T_{seq} = N^3.t_1 = O(N^3)$$

Parallel calculation for strips : $N.\frac{N}{P}$ elements , N op/el:

$$T_{cal} = \frac{N^3}{P}.t_1 = O(\frac{N^3}{P})$$

Parallel matrix multiplication: speedup using strips

Sequential calculation : $N.N$ elements, N operations/element:

$$T_{seq} = N^3.t_1 = O(N^3)$$

Parallel calculation for strips : $N.\frac{N}{P}$ elements , N op/el:

$$T_{cal} = \frac{N^3}{P}.t_1 = O(\frac{N^3}{P})$$

Parallel communication for strips : scatter A + broadcast B + gather C :

$$T_{com} = P.(2.\frac{N^2}{P} + N^2).t_2 \approx O(P.N^2)$$

Parallel matrix multiplication: speedup using strips

Sequential calculation : $N.N$ elements, N operations/element:

$$T_{seq} = N^3.t_1 = O(N^3)$$

Parallel calculation for strips : $N.\frac{N}{P}$ elements , N op/el:

$$T_{cal} = \frac{N^3}{P}.t_1 = O(\frac{N^3}{P})$$

Parallel communication for strips : scatter A + broadcast B + gather C :

$$T_{com} = P.(2.\frac{N^2}{P} + N^2).t_2 \approx O(P.N^2)$$

For $(N \gg P\frac{t_2}{t_1})$: $T_{par} = T_{cal} + T_{com} = O(\frac{N^3}{P}) + O(P.N^2) \approx O(\frac{N^3}{P})$

Parallel matrix multiplication: speedup using strips

Sequential calculation : $N.N$ elements, N operations/element:

$$T_{seq} = N^3.t_1 = O(N^3)$$

Parallel calculation for strips : $N.\frac{N}{P}$ elements , N op/el:

$$T_{cal} = \frac{N^3}{P}.t_1 = O(\frac{N^3}{P})$$

Parallel communication for strips : scatter A + broadcast B + gather C :

$$T_{com} = P.(2.\frac{N^2}{P} + N^2).t_2 \approx O(P.N^2)$$

For $(N \gg P\frac{t_2}{t_1})$: $T_{par} = T_{cal} + T_{com} = O(\frac{N^3}{P}) + O(P.N^2) \approx O(\frac{N^3}{P})$

Speedup : $\frac{T_{seq}}{T_{par}} = \frac{O(N^3)}{O(\frac{N^3}{P})} = O(P)$, speedup as long as: $P^2 < \frac{N.t_1}{t_2}$

Part 2: Hands on GPU, practical aspects

Control of GPU under CUDA and C: compilers and linkers

To make a Cuda program for the GPU we need some special statements in the code and extra options during compiling and linking.

CUDA provides its own compiler for C: nvcc. It compiles both normal CPU code (for which it simply uses the native C-compiler) and GPU code.

Control of GPU under CUDA and C: compilers and linkers

To make a Cuda program for the GPU we need some special statements in the code and extra options during compiling and linking.

CUDA provides its own compiler for C: nvcc. It compiles both normal CPU code (for which it simply uses the native C-compiler) and GPU code.

Some things to remember:

- Manually compile using : `% nvcc prg.c -o prg`
- Automatically compile by using e.g. a `Makefile`
- GPU code and CPU code can be in the same source (extension .cu)

Parallel “Hello World1” in Cuda C

A simple Hello World example in C.

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

#define NRBLKS   4 // Nr of blocks in a kernel (gridDim)
#define NRTPBK   4 // Nr of threads in a block (blockDim)

__global__ void printOnGPU()
{
   int myid = (blockIdx.x * blockDim.x) + threadIdx.x;
   printf("Hello World, I am thread %d \n",myid);
}

int main(void)
{
   printOnGPU <<< NRBLKS, NRTPBK >>> ();
   cudaDeviceReset(); // Without this line the output is NOT shown !
}
```

Parallel “Hello World2” in Cuda C

```
#include <stdio.h>
#include <cuda.h>
#define NRBLKS   4 // Nr of blocks in a kernel (gridDim)
#define NRTPBK   4 // Nr of threads in a block (blockDim)

__global__ void decodeOnGPU(char *string)
{ int myid = (blockIdx.x * blockDim.x) + threadIdx.x;
  string[myid] ^= (char)011; // 000001001 = 011 in octal
}
int main(void)
{  char *encryption = "Aleef)^f{em)((\11";
   char *string_h, * string_d; // pointers to host and device memory
   int len = strlen(encryption);

   string_h = (char *)malloc(     len);
   cudaMalloc((void **)&string_d, len);
   cudaMemcpy(string_d, encryption, len, cudaMemcpyHostToDevice);
   decodeOnGPU <<< NRBLKS, NRTPBK >>> (string_d);
   cudaMemcpy(string_h, string_d, len, cudaMemcpyDeviceToHost);

   printf("%s\n",string_h);
   cudaFree(string_d);
   free(string_h);
}
```

Unified Memory

A new feature since architecture 3.0 is Unified Memory. This creates a **single virtual memory space** for all GPU(s) and host(s) memory and automatically copies data if accessed from a place where it is not physically located. As such it makes `cudaMemcpy()` calls unnecessary, although the actual memory copies still need to be done under the hood.

Points of attention

- Keywords: `CudaMallocManaged()` or `__device__ __managed__`
- Can be slower than explicit memory management (useless transfers)...
- Must be compiled with architecture `sm_30` or higher
- Some Intel hardware MMUs may confuse this function, causing wrong answers without warnings. Boot with `iommu=soft` (on Linux)

Unified Memory

A new feature since architecture 3.0 is Unified Memory. This creates a **single virtual memory space** for all GPU(s) and host(s) memory and automatically copies data if accessed from a place where it is not physically located. As such it makes `cudaMemcpy()` calls unnecessary, although the actual memory copies still need to be done under the hood.

Points of attention

- Keywords: `CudaMallocManaged()` or `__device__ __managed__`
- Can be slower than explicit memory management (useless transfers)...
- Must be compiled with architecture `sm_30` or higher
- Some Intel hardware MMUs may confuse this function, causing wrong answers without warnings. Boot with `iommu=soft` (on Linux)

Example: Discuss and run `hello_unifiedmemory`

Parallel “Hello World2” in Cuda C (Unified Memory)

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

#define NRBLKS   4 // Nr of blocks in a kernel (gridDim)
#define NRTPBK   4 // Nr of threads in a block (blockDim)

__device__ __managed__ char string[] = "Aleef)^f{em)((\11";

__global__ void decodeOnGPU(char *string)
{ int myid = (blockIdx.x * blockDim.x) + threadIdx.x;
  string[myid] ^= (char)011; // 00001001 = 011 in octal
}
int main(void)
{
   decodeOnGPU <<< NRBLKS, NRTPBK >>> (string);
   cudaDeviceSynchronize();

   printf("%s\n",string);
}
```

Generic Cuda program framework

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

__global__ void routineOnGPU(float *data, int N)
{
   <GPU code>;
}
int main(void)
{
   float *data_h;  // pointer to host   memory
   float *data_d;  // pointer to device memory
   int size = N * sizeof(float);

   data_h = (float *)malloc(size);
   cudaMalloc((void **) &data_d, size);

   cudaMemcpy(data_d, data_h, size, cudaMemcpyHostToDevice);
   routineOnGPU <<< numberOfBlocks, threadsPerBlock >>> (data_d, N);
   cudaMemcpy(data_h, data_d, size, cudaMemcpyDeviceToHost);

   cudaFree(data_d);
   free(data_h);
}
```

Generic Cuda program framework with Unified Memory

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

__global__ void routineOnGPU(float *data, int N)
{
   <GPU code>;
}
int main(void)
{
   float *data;  // pointer to Unified host AND device memory
   int size = N * sizeof(float);

   cudaMallocManaged((void **) &data, size);

   routineOnGPU <<< numberOfBlocks, threadsPerBlock >>> (data, N);
   cudaDeviceSynchronize(); // or else no access to "data" from host

   cudaFree(data);
}
```

Another example: Scalar-Vector product (Multiply)

Example: Discussion of a Scalar-Vector product with various amounts of blocks. multiply1 uses simply 1 thread per block, multiply2 uses more.

Another example: Scalar-Vector product (Multiply)

Example: Discussion of a Scalar-Vector product with various amounts of blocks. multiply1 uses simply 1 thread per block, multiply2 uses more.

Test: Let's change the multiplication in `multiply3.cu` into the more time-consuming computation of $sin(\sqrt{a[i] * s})$ (changing the $t1/t2$ ratio) and recompile using `make`. Note that we use single precision to keep the results comparable: `sinf(sqrtf(a[i] * s))`

Another example: Scalar-Vector product (Multiply)

Example: Discussion of a Scalar-Vector product with various amounts of blocks. multiply1 uses simply 1 thread per block, multiply2 uses more.

Test: Let's change the multiplication in `multiply3.cu` into the more time-consuming computation of $sin(\sqrt{a[i] * s})$ (changing the $t1/t2$ ratio) and recompile using `make`. Note that we use single precision to keep the results comparable: `sinf(sqrtf(a[i] * s))`

Demo: Let's check for both multiply2 and 3 how the ratio between sequential and parallel changes for N=1000 (10^3), N=100000 (10^5) and N=10000000 (10^7) .

Scalar-Vector product (Multiply): K620 vs. Xeon 3.7 GHz

Problem size	N=10^3	N=10^5	N=10^7
Sequential multiply2 (ms)	2	20	3400
Parallel multiply2 (ms)	60	320	16000
ratio Parallel/Sequential	30	16	5

Scalar-Vector product (Multiply): K620 vs. Xeon 3.7 GHz

Problem size	N=10^3	N=10^5	N=10^7
Sequential multiply2 (ms)	2	20	3400
Parallel multiply2 (ms)	60	320	16000
ratio Parallel/Sequential	30	16	5

Problem size	N=10^3	N=10^5	N=10^7
Sequential multiply3 (ms)	25	1200	123000
Parallel multiply3 (ms)	74	330	16000
ratio Parallel/Sequential	3	0.27	0.13

Starting a kernel on the GPU

Recall the line from the previous examples:

```
routineOnGPU <<<numberOfBlocks, threadsPerBlock>>>(data_d);
```

Apart from this there are 2 extra and optional arguments:

- Reserve shared memory (often less important if caching is available).
- Associated stream in case more than 1 stream is used.

Starting a kernel on the GPU

Recall the line from the previous examples:

```
routineOnGPU <<<numberOfBlocks, threadsPerBlock>>>(data_d);
```

Apart from this there are 2 extra and optional arguments:

- Reserve shared memory (often less important if caching is available).
- Associated stream in case more than 1 stream is used.

Up to now we used 2 integers for number of blocks and threads per block. But generally these are of type `dim3` : a 3 dimensional vector with members `dim3.x`, `dim3.y` and `dim3.z`

Starting a kernel on the GPU

Recall the line from the previous examples:

```
routineOnGPU <<<numberOfBlocks, threadsPerBlock>>>(data_d);
```

Apart from this there are 2 extra and optional arguments:

- Reserve shared memory (often less important if caching is available).
- Associated stream in case more than 1 stream is used.

Up to now we used 2 integers for number of blocks and threads per block. But generally these are of type `dim3` : a 3 dimensional vector with members `dim3.x`, `dim3.y` and `dim3.z`

Max nr of blocks: $2^{31} - 1$ for x,y and z dims ($2^{16} - 1$ for x,y on old archs).
Max nr of threads: 1024 for x,y and 64 for z (512 for x,y on old archs).

CUDA: get device properties at runtime

To optimise a program for a certain GPU it is often required to obtain runtime information. This is done with `cudaGetDeviceProperties`:

```
int t,devcount;       // holds number of GPUs
cudaDeviceProp prop;  // holds GPU information

cudaGetDeviceCount(&devcount);

for(t=0;t < devcount; t++)
{ cudaGetDeviceProperties(&prop, t);
  fprintf(stderr,"memory  : %ld Bytes\n",prop.totalGlobalMem);
  fprintf(stderr,"major id: %d \n",prop.major);
  fprintf(stderr,"minor id: %d \n",prop.minor);
}
```

Example: Discussion of `DeviceInfo-GPU`

CUDA high level libraries: the CUBLAS library

Cuda also contains a BLAS library that works much like the popular **BLAS** library, but a great deal faster, especially for level 3 routines. Most of the gory GPU stuff is done inside the library and is something you don't have to think about.

Example: Discussion of `MatrixProd-Blas-GPU` (framework on next slide)

CUDA high level libraries: the CUBLAS library

Cuda also contains a BLAS library that works much like the popular **BLAS** library, but a great deal faster, especially for level 3 routines. Most of the gory GPU stuff is done inside the library and is something you don't have to think about.

Example: Discussion of `MatrixProd-Blas-GPU` (framework on next slide)

cublasSgemm

CUDA high level libraries: the CUBLAS library

Cuda also contains a BLAS library that works much like the popular **BLAS** library, but a great deal faster, especially for level 3 routines. Most of the gory GPU stuff is done inside the library and is something you don't have to think about.

Example: Discussion of `MatrixProd-Blas-GPU` (framework on next slide)

cublasSgemm

Demo: Show the CPU (Gsl version) and GPU programs for NxN matrices with N=1024, N=4096, N=2048, N=2047 and N=2049 and compare the results.

CUDA high level libraries: the CUBLAS library

Cuda also contains a BLAS library that works much like the popular **BLAS** library, but a great deal faster, especially for level 3 routines. Most of the gory GPU stuff is done inside the library and is something you don't have to think about.

Example: Discussion of `MatrixProd-Blas-GPU` (framework on next slide)

cublasSgemm

Demo: Show the CPU (Gsl version) and GPU programs for NxN matrices with N=1024, N=4096, N=2048, N=2047 and N=2049 and compare the results.

Note: The difference between 2047 and 2048 can be almost a factor 4 on older GPU architectures! However, on 2.x and up and on CPU there should hardly be any difference.

Generic Cuda Blas framework Sgemm

```
#include <stdio.h>
#include <cuda.h>
#include <cublas_v2.h>
int main(void)
{
   float *data_hA, *data_hB, *data_hC;  // pointers to host   memory
   float *data_dA, *data_dB, *data_dC;  // pointers to device memory
   cublasHandle_t handle;
   int size = dim * dim * sizeof(float);

   cublasCreate(&handle);
   data_hA = (float *)malloc(size);      // 3 times for A, B and C
   cudaMalloc((void **) &data_dA, size); // 3 times for A, B and C

   cublasSetVector(dim*dim, sizeof(float), data_hA, 1, data_dA, 1); // 2x : A and B
   status = cublasSgemm(handle, ... data_dA, dim, data_dB, dim, ..., data_dC, dim);
   cublasGetVector(dim*dim, sizeof(float), data_dC, 1, data_hC, 1);

   cublasDestroy(handle);
   cudaFree(data_dA); // 3 times for A, B and C
   free(data_hA);     // 3 times for A, B and C
}
```

Generic Cuda Blas framework Sgemv

```
#include <cuda.h>
#include <cublas_v2.h>
int main(void)
{
   float *h_A, *h_x, *h_b;  // matrix A and vectors b,x in host memory
   float *d_A, *d_x, *d_b;  // matrix A and vectors b,x in device memory
   float alpha = 1, beta = 0;
   cublasHandle_t handle;

   cublasCreate(&handle);
   cudaMallocHost(&h_A, dim * dim * sizeof(float));
   cudaMallocHost(&h_x, dim *       sizeof(float)); // and once more for b
   cudaMalloc    (&d_A, dim * dim * sizeof(float));
   cudaMalloc    (&d_x, dim *       sizeof(float)); // and once more for b

   cublasSetVector(dim*dim, sizeof(float), h_A, 1, d_A, 1);
   cublasSetVector(dim    , sizeof(float), h_x, 1, d_x, 1);
   status = cublasSgemv(handle, ... , &alpha, d_A, dim, d_x, 1, &beta, d_b, 1);
   cublasGetVector(dim    , sizeof(float), d_b, 1, h_b, 1);

   cublasDestroy(handle);
   cudaFree    (d_A); cudaFree    (d_x); cudaFree    (d_b);
   cudaFreeHost(h_A); cudaFreeHost(h_x); cudaFreeHost(h_b);
}
```

Generic Cuda Blas framework Sgemv (unified memory)

```
#include <cuda.h>
#include <cublas_v2.h>
int main(void)
{
   float *h_A, *h_x, *h_b;  // matrix A and vectors b,x in managed memory
   float alpha = 1, beta = 0;
   cublasHandle_t handle;

   cublasCreate(&handle);
   cudaMallocManaged(&h_A, dim * dim * sizeof(float));
   cudaMallocManaged(&h_x, dim *       sizeof(float)); // and once more for b

   status = cublasSgemv(handle, ... , &alpha, h_A, dim, h_x, 1, &beta, h_b, 1);
   cudaDeviceSynchronize();  // must wait until result can be used !

   cublasDestroy(handle);
   cudaFree(h_A); cudaFree(h_x); cudaFree(h_b);
}
```

Debugging of memory leak errors using cuda-memcheck

cuda-memcheck is a memory leak checker for CUDA and works much the same as the Unix tool valgrind. Note that it is replaced since cuda 11 with compute-sanitizer, which however works in exactly the same way.

Notes

- Start it with: `% compute-sanitizer <yourprogram>`
- Compute-sanitizer always stops after the first memory error. You can force it to continue for the rest of the program using:
  `% compute-sanitizer --destroy-on-device-error kernel <yourprogram>`

Debugging of memory leak errors using cuda-memcheck

cuda-memcheck is a memory leak checker for CUDA and works much the same as the Unix tool valgrind. Note that it is replaced since cuda 11 with compute-sanitizer, which however works in exactly the same way.

Notes

- Start it with: `% compute-sanitizer <yourprogram>`
- Compute-sanitizer always stops after the first memory error. You can force it to continue for the rest of the program using:
  `% compute-sanitizer --destroy-on-device-error kernel <yourprogram>`

Demo: Let's check the Memcheck-GPU example using `compute-sanitizer` and see if you can find what is wrong in the code there?

Profiling Cuda code

Since Cuda 5 there is a commandline profiler nvprof, now being replaced (Cuda 11) with nsys nvprof. Use as: `nvprof myprogram <arguments>`

There are also GUI profilers named nvvp since Cuda 4 and one named nsys-ui since Cuda 11 that replaces it.

Example: `matrixprod-simple` with 2 kernels using nvprof (arch $<=$ 7.x).

Profiling Cuda code

Since Cuda 5 there is a commandline profiler nvprof, now being replaced (Cuda 11) with nsys nvprof. Use as: `nvprof myprogram <arguments>`

There are also GUI profilers named nvvp since Cuda 4 and one named nsys-ui since Cuda 11 that replaces it.

Example: `matrixprod-simple` with 2 kernels using nvprof (arch $<=$ 7.x).

Example: `matrixprod-simple` using nsight-sys (nsys-ui) (cuda $>=$ 11.x).

Profiling Cuda code

Since Cuda 5 there is a commandline profiler nvprof, now being replaced (Cuda 11) with nsys nvprof. Use as: `nvprof myprogram <arguments>`

There are also GUI profilers named nvvp since Cuda 4 and one named nsys-ui since Cuda 11 that replaces it.

Example: `matrixprod-simple` with 2 kernels using nvprof (arch $<=$ 7.x).

Example: `matrixprod-simple` using nsight-sys (nsys-ui) (cuda $>=$ 11.x).

Example: Show `parbody-psm1` (1024grid.par) with the nsight-sys GUI.

Profiling Cuda code

Since Cuda 5 there is a commandline profiler nvprof, now being replaced (Cuda 11) with nsys nvprof. Use as: `nvprof myprogram <arguments>`

There are also GUI profilers named nvvp since Cuda 4 and one named nsys-ui since Cuda 11 that replaces it.

Example: `matrixprod-simple` with 2 kernels using nvprof (arch <= 7.x).

Example: `matrixprod-simple` using nsight-sys (nsys-ui) (cuda >= 11.x).

Example: Show `parbody-psm1` (1024grid.par) with the nsight-sys GUI.

Demo: Show nvprof and possibly nsys-ui on a modern system (.arch <= 7.x).

Note: Select localhost connection, set commandline with arguments (full path!), enable Collect CUDA trace and press Start. Select Timeline View and expand the CUDA HW section to view kernel statistics like in nvvp.

Use zoom if required (select region, press right mouse and choose Zoom into selection).

Some nice examples

Let's try a few nice (graphic) examples, such as:

- fluidsGL
- smokeParticles
- particles
- Mandelbrot

Part 3: Just a simple exercise

Exercise

Example: Let's first shortly discuss the example `MatVecProd-Blas-GPU`, where we compute a matrix vector product using the CuBlas library. So, no need to write your own kernels;-)

Exercise

Example: Let's first shortly discuss the example `MatVecProd-Blas-GPU`, where we compute a matrix vector product using the CuBlas library. So, no need to write your own kernels;-)

Exercise: In the directory `MatrixProd-Blas-GPU-exercise` you will find **exactly** the same code as in `MatVecProd-Blas-GPU`, only the names are changed into a matrix matrix product. Your task is to try to modify this example in such a way that it computes a real Matrix Product using CuBlas on the GPU!

Note: Maybe the best option is to use the one with explicit memory transfers as there it is much easier to time the actual computation. With unified memory - albeit slightly easier - it is much harder to avoid measuring unexpected implicit and hidden host <-> card transfers as well.

Exercise

Example: Let's first shortly discuss the example `MatVecProd-Blas-GPU`, where we compute a matrix vector product using the CuBlas library. So, no need to write your own kernels;-)

Exercise: In the directory `MatrixProd-Blas-GPU-exercise` you will find **exactly** the same code as in `MatVecProd-Blas-GPU`, only the names are changed into a matrix matrix product. Your task is to try to modify this example in such a way that it computes a real Matrix Product using CuBlas on the GPU!

Note: Maybe the best option is to use the one with explicit memory transfers as there it is much easier to time the actual computation. With unified memory - albeit slightly easier - it is much harder to avoid measuring unexpected implicit and hidden host <-> card transfers as well.

Discuss shortly the solution.

THE END

Thanks for your attention!