GPU Programming
or
How to tremendously accelerate your code?

Michael Kraus, Christian Konz

Max-Planck-Institut für Plasmaphysik, Garching

Ringberg Theorie Seminar 2010

GPU Programming? GPU?

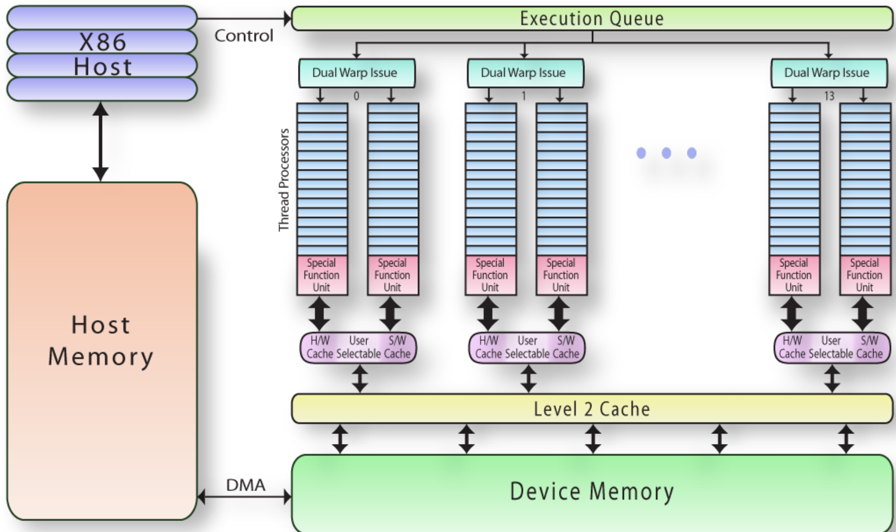
- GPUs can do more than just render nice scenes in pc games
- highly parallel, multithreaded, manycore processors (448 cores)
- tremendous computational power (515 gigaflop/s double prec)
- very high memory bandwidth (144 GB/s)
- relatively cheap (board: ∼€2.500, 4 GPU unit: ∼€10.000)
- low power consumption (board: 238W, 4 GPU unit: 900W)
- possible performance gain of 10 ∼ 20x, sometimes 100 ∼ 200x

- comparison: performance per $
  - 2x Xeon X5550 (Nehalem) 2.66GHz, 48GB, $7K, 0.55KW, 80 Gflop/s
  - 2x Xeon X5550 + 2x Tesla C2050, 48GB, $11K, 1.0KW, 656 Gflop/s

- Nebulae (National Supercomputing Centre in Shenzhen, China):
  - #2 in June's Top500 list
  - record performance of 1.75 petaflop/s (Linpack: 1.2)
  - theoretical peak performance of 2.3 petaflop/s
  - 9280 Intel Xeon X5650 CPUs with 6 cores each
  - 4640 NVidia Tesla C2050 GPUs

Tianhe-1A
#1 Nov'10 (?)
2.5 petaflop/s

GPU Architecture



GPU Architecture

- hardware outline:
  - 4 clusters, each containing 4 streaming processors with 32 cores
  - each cluster can do 16 double precision operations per cycle
  - each streaming processor contains 64kb cache memory
  - special function units: cos, sin
- memory hierarchy:
  - global memory (onboard, DRAM): per application
  - shared memory (onchip): per block
  - private memory (placed in registers or local memory): per thread
- thread hierarchy: grids $\rightarrow$ blocks $\rightarrow$ threads
  - threads are organised (by the programmer) in grids of thread blocks
  - thread block: set of concurrent threads (max. 512), that can cooperate among themselves through barrier synchronisation and shared memory private to the block
  - grid: set of thread blocks that may each be executed independently and thus may execute in parallel
  - grids and blocks may have one, two or three dimensions

GPU Architecture

- well suited for data-parallel computations (the same algorithm/ calculation is executed on many data elements in parallel)
- data elements get mapped to parallel processing threads
- efficient parallelisation: assign one thread to each output element
  - ⇒ each iteration is independent of all others
  - ⇒ no need for synchronisation among threads when writing results to memory
- host and devices have separate memory spaces
  - ⇒ to use CUDA, data values must be transferred from the host to the device along the PCI bus (costly, should be minimised)
- a compiled CUDA program executes on any number of processors, parallel execution and thread management are automatic
  - ⇒ thread creation, scheduling and termination are handled by the underlying system
  - ⇒ Tesla and Fermi GPUs perform all thread management directly on hardware with almost no overhead (e.g. create 32 threads per cycle)

NVIDIA Tesla/Fermi

- NVidia Tesla (available since 2007/2008):
  - single precision: 933 gigaflop/s peak performance
  - double precision: 78 gigaflop/s peak performance
  - 240 computing cores, max. 4 GB RAM
- NVidia Fermi (available since early 2010):
  - single precision: 1030 gigaflop/s peak performance
  - double precision: 515 gigaflop/s peak performance
  - 448 computing cores, max. 6 GB RAM, ECC memory error protection
  - optimised for HPC applications
- RZG:
  - one NVIDIA S1070 unit with 4 Tesla GPUs
  - two CPU servers, each equipped 2 Intel Nehalem quadcores
  - each server is connected to 2 GPUs
  - soon: additional NVIDIA S2070 unit with 4 Fermi GPUs
  - then: each GPU unit connected to one server

Programming Models and Languages

- OpenCL (low-level)
  - open standard (Khronos Group: Apple, Intel, AMD/ATI, NVIDIA, ...)
  - platform/vendor independent (supported by GPUs from NVIDIA and AMD/ATI, Cell processors, Intel Larrabee, ...)
  - supports only C and C++ (but free)
- CUDA (intermediate-level)
  - defined by NVIDIA, only available on their devices
  - works on HPC cards as well as your common graphics card
  - natively supports C and C++ (NVIDIA CUDA Toolkit, free)
  - Fortran: Portland Compiler ($$$)
  - Python Module (pyCUDA, free)
  - free Libraries: FFTW, BLAS, sparse matrix routines
  - commercial Libraries: CULA LAPACK (Fortran, C, C++)
- PGI Accelerator (Portland, high-level)
  - similar to OpenMP (even easier!)
  - supports Fortran, C, C++

Memory Access

- variables are allocated on host and device (main program)

```
real :: A(M,N)                ! A instantiated in host memory
real, device :: Adev(M,N) ! Adev instantiated in GPU memory
```

- data has to be copied between host and device

```
Adev = A                      ! Copy data from A (host) to Adev (GPU)
 ...
A = Adev                      ! Copy data from Adev (GPU) to A (host)
```

- specify shared and private memory (within a kernel):

```
real :: B(M,N)                ! B instantiated in private memory (thread)
real, shared :: C(M,N)        ! C instantiated in shared  memory (block)
```

- in C this is slightly more complicated:

```
cudaMemcpy(Adev, A, M*N, cudaMemcpyHostToDevice);
cudaMemcpy(A, Adev, M*N, cudaMemcpyDeviceToHost);
```

CUDA Kernels (Matrix Multiplication)

- kernel: a function compiled for and executed on the device
- the kernel is executed on the device by many different threads
- define a kernel for your problem:

```
attributes(global) subroutine MMUL_KERNEL(A,B,C,N,M,L)
  ...
  tx = threadidx%x ; ty = threadidx%y
  i = (blockidx%x-1) * 16 + tx
  j = (blockidx%y-1) * 16 + ty
  C(i,j) = 0.0
  do kb = 1, M, 16
     do k = 1, 16
        C(i,j) = C(i,j) + A(i,kb+ty-1) * B(kb+tx-1,j)
     enddo
  enddo
end subroutine
```

- call it from your program:

```
call mmul_kernel<<<dimGrid,dimBlock>>>(Adev,Bdev,Cdev,N,M,L)
```

PGI Accelerator (Matrix Multiplication)

- PGI Accelerator Compiler:

```
!$ACC REGION
   do k = 1,n1
      do i = 1,n3
         c(i,k) = 0.0
         do j = 1,n2 c(i,k) = c(i,k) + a(i,j) * b(j,k)
      enddo
   enddo
!$ACC END REGION
```

- OpenMP:

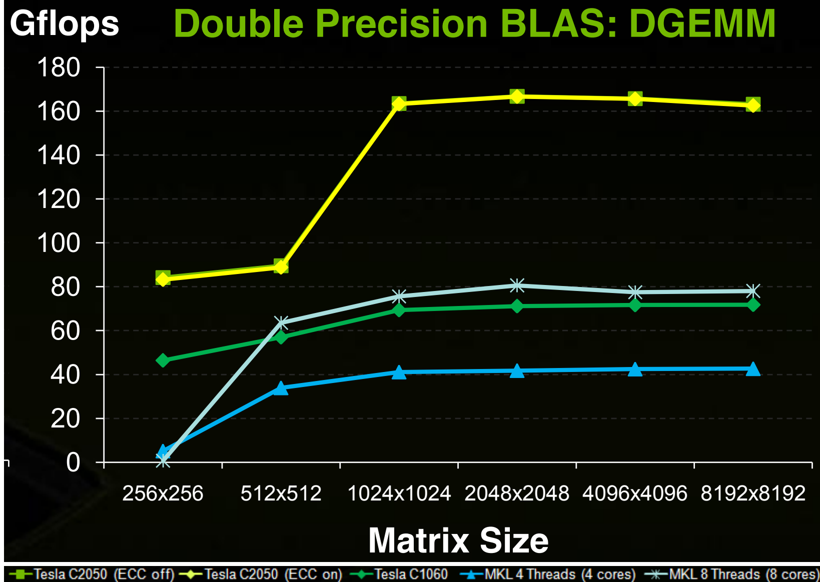
```
!$OMP PARALLEL DO SHARED(a,b,c,n,n2,n3) PRIVATE(i,j,k)
   do k = 1,n1
      do i = 1,n3
         c(i,k) = 0.0
         do j = 1,n2 c(i,k) = c(i,k) + a(i,j) * b(j,k)
      enddo
   enddo
!$OMP END PARALLEL DO
```

Where to use GPUs? Where to benefit?

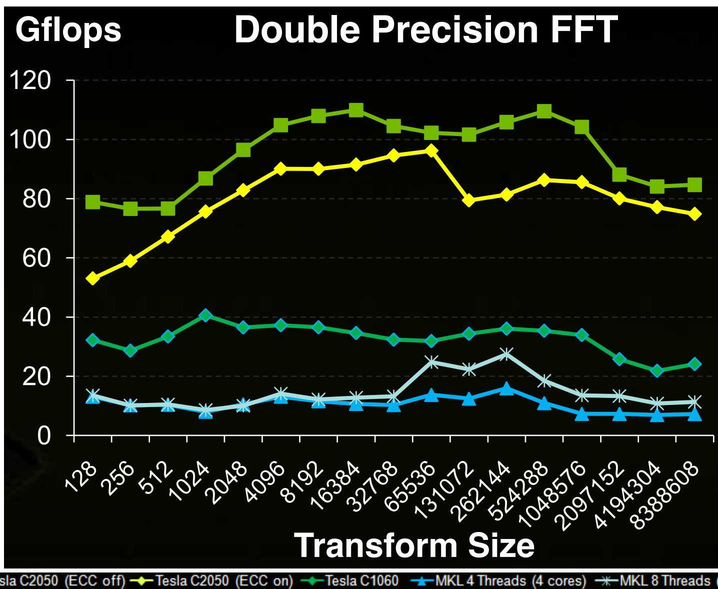
- use GPU version of external libraries like BLAS, FFT, LAPACK

  ⇒ easily gain a factor of $2 \sim 10$ by just calling a different library
  - example: trigonometric functions (double precision)
    - 5x faster on Tesla C2050 than on Intel i7 980X hexacore
  - example: NVIDIA's CUBLAS 3.1 compared to MKL 10.2
    - up to 4x faster on Tesla C2050 than on Xeon 5550 quadcore (but: problem size very important!)
  - example: NVIDIA's CUFFT 3.1 compared to MKL 10.2
    - $5 \sim 10$x faster on Tesla C2050 than on Xeon 5550 quadcore
  - example: CULA LAPACK compared to MKL
    - $2 \sim 7$x faster on Tesla C2050 than on Intel i7 quadcore

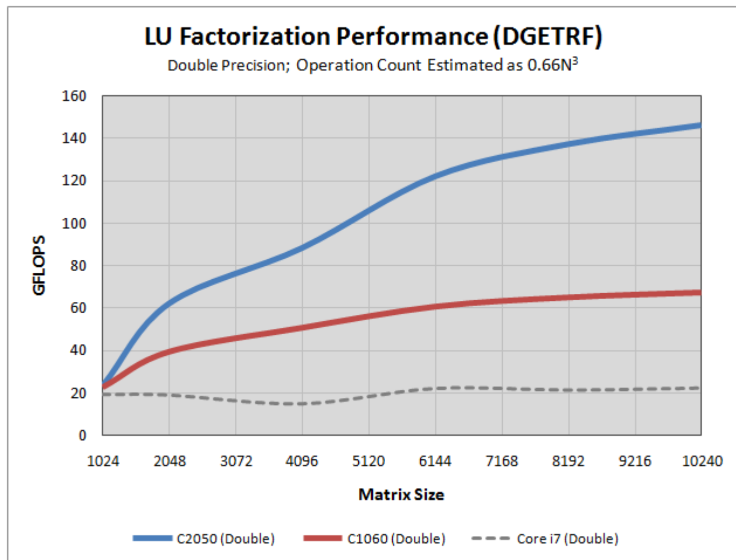
CUBLAS 3.1 vs. Intel MKL 10.2 (Xeon 5550 quad)



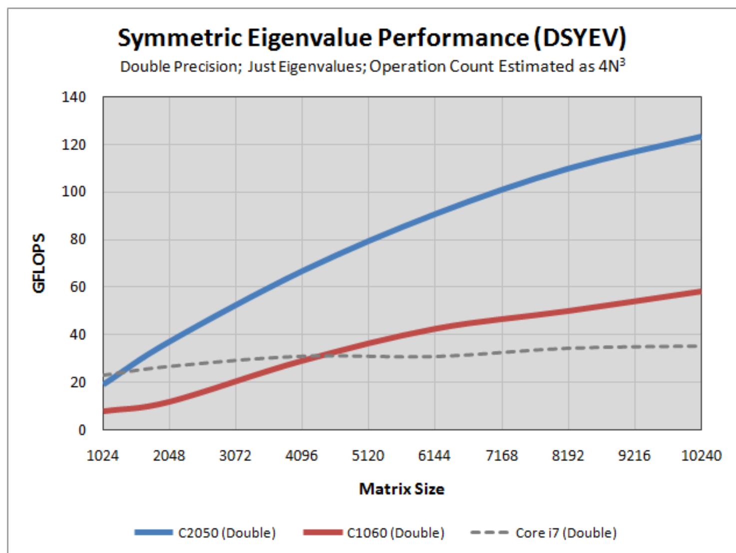
CUFFT 3.1 vs. Intel MKL 10.2 (Xeon 5550 quad)



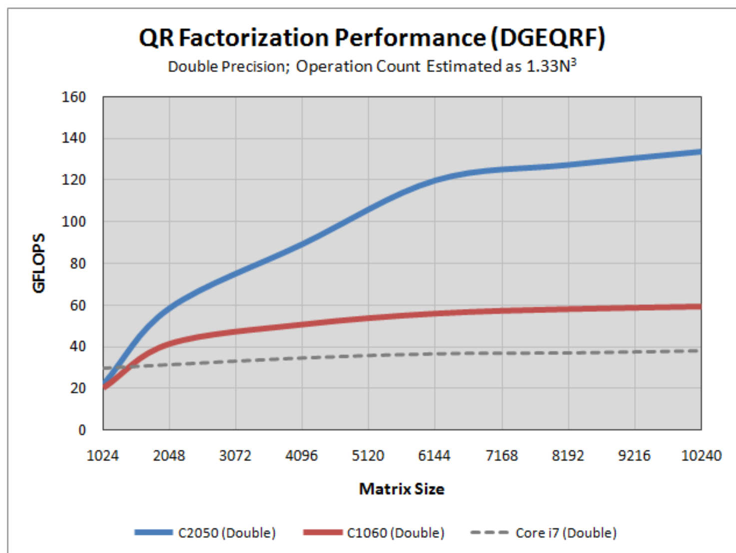
CULA 2.0 vs. MKL 10.2 (Linear Equations)



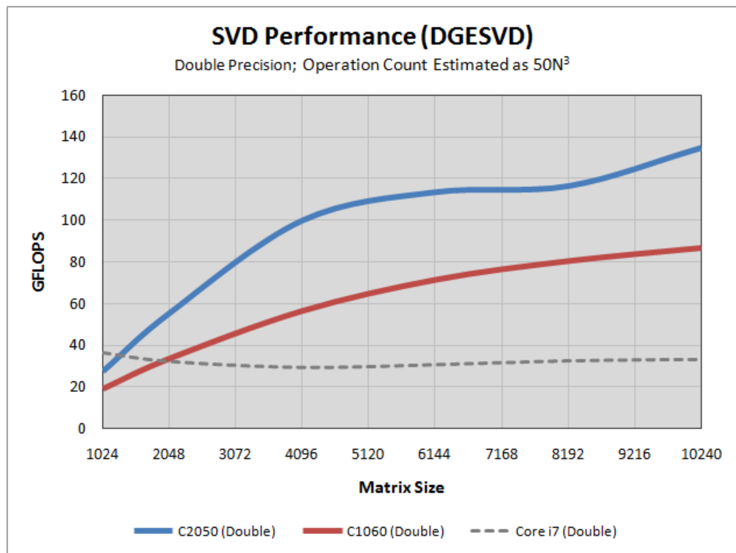
CULA 2.0 vs. MKL 10.2 (Symmetric Eigenproblems)



CULA 2.0 vs. MKL 10.2 (Orthogonal Factorisations)



CULA 2.0 vs. MKL 10.2 (Singular Value Decomp)



Where to use GPUs? Where to benefit?

- use GPU versions of external libraries like BLAS, FFT, LAPACK

  $\Rightarrow$ easily gain a factor of $2 \sim 10$ by just calling a different library
  - example: trigonometric functions (double precision)
    - 5x faster on Tesla C2050 than on Intel i7 980X hexacore
  - example: NVIDIA's CUBLAS 3.1 compared to MKL 10.2
    - up to 4x faster on Tesla C2050 than on Xeon 5550 quadcore (but: problem size very important!)
  - example: NVIDIA's CUFFT 3.1 compared to MKL 10.2
    - $5 \sim 10$x faster on Tesla C2050 than on Xeon 5550 quadcore
  - example: CULA LAPACK compared to MKL
    - $2 \sim 7$x faster on Tesla C2050 than on Intel i7 quadcore
- bottle-neck: large latency in CPU-GPU communication

  $\Rightarrow$ you need either lots of data which can be processed in parallel

  $\Rightarrow$ or you should perform lots of (independent) operations on your data
- ideal case: inherent high level of parallelism in your application

  $\Rightarrow$ optimal exploitation of the many cores
  - PIC, Monte Carlo, ...

Drawbacks, Caveats, Open Questions

⇒ **portability**:
- CUDA only available on NVIDIA devices, but NVIDIA is porting it to x86 devices right now
- OpenCL is multi-architecture, but only C/C++ and slightly more difficult to program; as yet robust support only on NVIDIA devices
- PGI compilers are expensive but ease programming (but proprietary)

⇒ but the OpenMP language committee has a subgroup working on standardising a set of accelerator directives, to allow programming and tuning across a wide range of accelerators

- in CUDA you can easily mess up memory
- performance can heavily depend on number of threads
- copying data between host and GPU is quite slow
- global memory has a high latency (400-600 cycles), but when a thread hits a memory read or write instruction, the scheduler is able to run arithmetic instructions from other threads

Summary, Interesting Projects

- possibility of speeding up your code with various amounts of effort
- usefulness heavily depending on the type of code/problem
- right now: probably mainly interesting to speed up small-scale codes and/or data analysis routines
- but: GPUs and CUDA or similar devices and programming models will probably be used in future super computers
  - ⇒ good to have some experience in programming such devices
- existing projects at IPP:
  - Grad-Shafranov solver for real-time equilibrium reconstruction (master thesis supervised by U. von Toussaint)
- possible projects:
  - GPU version of interpolation tools (Dierckx library)
  - non-linear cartesian MHD code (C++) for reconnection studies
  - GPU solver for ILSA (depending on BLAS, LAPACK)

⇒ Looking for your ideas!

⇒ Question: Buy the Portland Fortran Compiler? (or rather convince the RZG to do so)

Future Developments and Alternatives

- IBM Cell processors:
  - IBM PowerXCell 8i processors with 3,2 GHz: 1.5 teraflop/s double precision performance
  - IBM XL compilers for Fortran and C/C++
  - availability of libraries similar to GPU: BLAS, FFT, LAPACK, ...
- Intel's Many Core Architecture: Knights Ferry & Knights Corner
  - Knights Ferry has 32 x86 cores on chip, allowing 4 threads per core
  - each core has a vector unit, allowing 16 single precision or 8 double-precision floating point operations in a single instruction
  - implemented on a PCI card with own memory, connected to the host memory through PCI DMA operations
  - Intel Fortran and C/C++ compilers, math libraries, etc.
  - Knights Corner will be available mid-2011 and contain > 50 cores
- AMD Llano (mid-2011)
  - integration of CPU and GPU on one die
  - 4 x86 Cores, 480 Stream Processors
  - rather consumer oriented?

References

- Im Zeitalter des CUDAismus (M. Fischer, A. Stiller, c't 19/2010)
- Scalable Parallel Programming (J. Nickolls et al, ACM Queue '08)
- NVIDIA High Performance Computing:
  `www.nvidia.com/object/tesla_computing_solutions.html`
- Introduction to PGI CUDA Fortran (PGI Insider):
  `www.pgroup.com/lit/articles/insider/v1n3a2.htm`
- The PGI Accelerator Programming Model on NVIDIA GPUs (Michael Wolfe, PGI Insider):
  `www.pgroup.com/lit/articles/insider/v1n1a1.htm`
- RZG NVIDIA Tesla S1070:
  `www.rzg.mpg.de/computing/hardware/miscellaneous/nvidia-tesla-s1070`
- CULA LAPACK:
  `www.culatools.com`

Portland Compiler: Pricing

PGI Accelerator x64+GPU for Linux:

	License	Subscription
Workstation: Fortran	\$1.059	\$350
Workstation: Fortran/C/C++	\$1.499	\$500
Server (2 User): Fortran	\$4.239	\$1.170
Server (2 User): Fortran/C/C++	\$5.999	\$1.650
Server (5 User): Fortran	\$7.949	\$2.190
Server (5 User): Fortran/C/C++	\$11.249	\$3.090

also supports OpenMP, allowing you to mix GPU and CPU parallelisation