Using PGI compilers to solve initial value problems on GPU

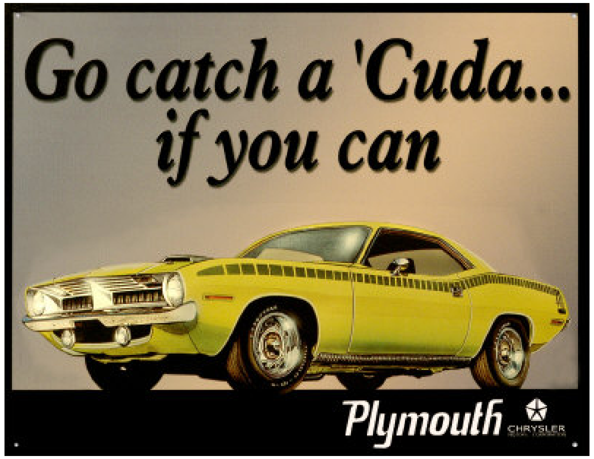
Ladislav Hanyk
Charles University in Prague, Faculty of Mathematics and Physics
Czech Republic

Outline

1. NVIDIA GPUs and CUDA developer tools
2. PGI CUDA Fortran
3. PGI and OpenACC directives
4. Initial value solvers with PGI compilers
5. PGI compilers on the Cascade system, a performance benchmark

1. NVIDIA GPUs

CUDA: Compute Unified Device Architecture



Cuda in 1970s

– the most popular GP GPU parallel programming model today

– from notebooks and personal desktops
to high performance computing

– a host (CPU) offloads a suitable part of a process (a kernel)
to the device (GPU)

– the device with many cores runs the kernel concurrently
by many subprocesses (threads)

– two-level hardware parallelism on a device:
SIMD (single-instruction multiple-data) and
MIMD (multiple-instruction multiple-data)

– a programming model reflects the hardware parallelism by grouping the threads into blocks and grids

1. NVIDIA GPUs

Product generations (architectures) and compute capability

G80 (since 2006): compute capability 1.0, 1.1

features 1.1: 8 cores/multiprocessor, single-precision (SP) real arithmetic
models: GeForce 9800, Quadro FX 5600, Tesla C870, D870, S870

GT200 (since 2008): compute capability 1.2, 1.3

features 1.3: 8 cores/multiprocessor, max. 30 x 8 = 240 cores, double-precision (DP)
models: GeForce GTX 285/295, Quadro FX 5800, Tesla C1060, M1060, S1070

Fermi (since 2010): compute capability 2.0, 2.1

features 2.0: 32 cores/multiprocessor, max. 16 x 32 = 512 cores, fast DP, cache
2.1: 48 cores/multiprocessor, max. 8 x 48 = 384 cores, slower DP
models: GeForce GTX 580/590, Quadro 6000, Tesla C2075, M2090, S2070

Kepler (since 2012): compute capability 3.0

features 3.0: 192 cores/multiprocessor, max. 8 x 192 = 1536 cores, slower DP
models: GeForce GTX 680/690

All CUDA-capable GPUs: http://developer.nvidia.com/cuda-gpus

Product lines (classes)

GeForce for games and PC graphics
Quadro for professional graphics
Tesla for HPC

1. CUDA developer tools

CUDA – a general purpose parallel computing architecture
hardware: multiprocessor, cores, memory hierarchy
software: a programming model with grids, blocks, warps, threads, memory limits etc.
C/C++ compiler nvcc and API library
CUDA Toolkit with nvcc, debugger and Visual Profiler
GPU-accelerated numerical libraries: CUBLAS, CUSPARSE, CUFFT, CURAND
Computing SDK (Software Development Kit) code samples

More tools by third parties:
OpenCL (Khronos), Brook (Stanford University) – based on C language
Microsoft DirectCompute – a part of DirectX
Jacket (AccelerEyes) – platform for MATLAB

From now on:
PGI (The Portland Group) Compiler Suite
– with PGI CUDA Fortran extensions
– with PGI Accelerator and OpenACC directives

2. PGI compiler suite

PGI Workstation, PGI Server, PGI CUDA Fortran, PGI CUDA-x86, PGI Accelerator
recent version 12.4 (4th minor version in 2012,
10 minor versions in 2011)
for Linux, Mac OS X and Microsoft Windows, 32/64 bit
with OpenMP, MPI, parallel debugger and profiler
with ACML and IMSL libraries (and linkable to Intel MKL)
with Eclipse (Linux) or Microsoft Visual Studio (Windows)

Mt. Hood, Oregon

Support for GPU computing

PGI CUDA Fortran extensions (since 2010)
PGI Accelerator directives (like OpenMP, since 2010)
OpenACC directives: a new, standardized specification of directives (since 2012)
PGI CUDA-x86: emulation of Fortran or C with CUDA extensions on CPUs
NVIDIA compiler and libraries included (cublas, cusparse, cufft, curand)

PGI home
http://www.pgroup.com

2. PGI CUDA Fortran mission

In short: a low-level approach, similar to nvcc by NVIDIA (with a few substantial enhancements)
kernels explicitly written and launched
explicit global and local GPU memory management
value added: simple data transfers & kernel loop directive to generate kernels automatically

A closer look:

– writing kernel subroutines and device procedures: procedure attributes

```
ATTRIBUTES(GLOBAL) SUBROUTINE MyKernel(arguments)
ATTRIBUTES(DEVICE) FUNCTION MyDeviceFunction(arguments)
```

– declaring and allocating data on GPU: data attributes

```
INTEGER,ALLOCATABLE,DEVICE :: ad(:) ! device memory
REAL :: b                           ! registers or local memory
REAL,CONSTANT :: pi                 ! constant memory
COMPLEX,SHARED :: c(nmax)           ! shared memory
```

– transferring data between host and device: assignment statements

```
ad=a ; ... ; a=ad
```

– launching kernels: chevron syntax

```
call MyKernel<<<gridsize,blocksize>>>(arguments)
```

– calling CUDA Runtime API routines: CUDA API functions

```
istatus=cudaThreadSynchronize()
```

– accessing CUDA types, constants, variables and interfaces: module cudafor

```
use cudafor
```

2. Refactoring source codes for CUDA Fortran: To-do list

1. extract the parallelizable code into subroutines contained in a module

2. edit the kernel:
   set the global attribute of the kernel
   set the value attributes of kernel arguments passed by value
   substitute outer loops with thread indexing

3. edit the host procedure:
   attach the cudafor module
   set grid and block shape and sizes
   allocate device memory data
   transfer data from host memory to device memory
   set the execution configuration by chevrons
   pass kernel arguments: arrays in device memory by reference, scalars in host memory by value
   transfer data from device memory back to host memory

4. compile and debug:
   pgfortran -fast -Mcuda -Minfo file.f90

2. Example with PGI CUDA Fortran: array update, a(:)=a(:)+z

a CPU version ... pgfortran -fast t-cpu.f90

```
MODULE m

IMPLICIT NONE
INTEGER,PARAMETER :: DP=4,NMAX=4096*256


CONTAINS

SUBROUTINE Update(a,z)
REAL(DP) :: a(:)
REAL(DP) :: z
INTEGER  :: j
do j=1,size(a)
  a(j)=a(j)+z
enddo
END SUBROUTINE

END MODULE

PROGRAM Test_CPU
USE m
IMPLICIT NONE
REAL(DP) :: a(NMAX),z


a=0.
z=1.
call Update(a,z)

print *,a(1),sum(a)

END PROGRAM
```

a GPU version ... pgfortran -fast -Mcuda t-pgi-cuf.f90

```
MODULE m
USE cudafor
IMPLICIT NONE
INTEGER,PARAMETER :: DP=4,NG=4096,NB=256,NMAX=NG*NB
TYPE(dim3) :: grid=dim3(NG,1,1),block=dim3(NB,1,1)

CONTAINS

ATTRIBUTES(GLOBAL) SUBROUTINE Update(a,z)
REAL(DP),DEVICE :: a(:)
REAL(DP),VALUE :: z
INTEGER  :: j
  j=threadidx%x+NB*(blockidx%x-1)
  a(j)=a(j)+z

END SUBROUTINE

END MODULE

PROGRAM Test_GPU
USE m
IMPLICIT NONE
REAL(DP) :: a(NMAX),z
REAL(DP),DEVICE :: ad(NMAX)

ad=0.
z=1.
call Update<<<grid,block>>>(ad,z)
a=ad
print *,a(1),sum(a)

END PROGRAM
```

2. PGI CUDA Fortran: Kernel loop directive

Kernel loop directive

– to transform loops into kernels automatically
– to establish the correspondence between loop bounds and grid/block sizes automatically
– to unroll selected iterations for a sequential run on GPU

`!$CUF KERNEL DO <<<grid,block>>>`	for a standalone (or outer) loop
`!$CUF KERNEL DO(n) <<<grid,block>>>`	for tightly-nested loops

where n is a number of tightly-nested loops
and grid and block are `*`, scalar integers or a parenthesized list of `*` or integers

Examples of executable configurations:

`<<<*,*>>>`	compiler's choice of grid/block sizes
`<<<*,256>>>`	explicit blocksize, gridsize = loopsize / blocksize derived by compiler
`<<<1,256>>>`	explicit grid/block size, excess iterations run sequentially in each thread
`<<<*,(256,1)>>>`	for nested loops: explicit 1D blocksize, 2D gridsize derived by compiler
`<<<(1,*),(256,1)>>>`	all in one: explicit blocksize with unrolled inner iterations, gridsize derived

More in PGI CUDA Fortran Programming Guide.

Care needed to preserve device-memory coalescing:
consecutive threads should update consecutive array elements.

2. Example with PGI CUDA Fortran and a kernel loop directive: array update, a(:)=a(:)+z

a CPU version ... pgfortran -fast t-cpu.f90

```
MODULE m
IMPLICIT NONE
INTEGER,PARAMETER :: DP=4,NMAX=4096*256

CONTAINS

SUBROUTINE Update(a,z)
REAL(DP) :: a(:)
REAL(DP) :: z
INTEGER  :: j



do j=1,size(a)
  a(j)=a(j)+z
enddo

END SUBROUTINE

END MODULE

PROGRAM Test_CPU
USE m
IMPLICIT NONE
REAL(DP) :: a(NMAX),z

a=0.
z=1.
call Update(a,z)
print *,a(1),sum(a)

END PROGRAM
```

a GPU version ... pgfortran -fast -Mcuda t-pgi-cuf-kernel.f90

```
MODULE m
IMPLICIT NONE
INTEGER,PARAMETER :: DP=4,NG=4096,NB=256,NMAX=NG*NB

CONTAINS

SUBROUTINE Update(a,z)
REAL(DP) :: a(:)
REAL(DP) :: z
INTEGER  :: j
REAL(DP),DEVICE :: ad(NMAX)
ad=a
!$CUF KERNEL DO <<<NG,NB>>>
do j=1,size(a)
  ad(j)=ad(j)+z
enddo
a=ad
END SUBROUTINE

END MODULE

PROGRAM Test_GPU
USE m
IMPLICIT NONE
REAL(DP) :: a(NMAX),z

a=0.
z=1.
call Update(a,z)
print *,a(1),sum(a)

END PROGRAM
```

3. PGI Accelerator directives

In short: the high-level approach for offloading codes to an accelerator
similar to OpenMP directives for thread parallelization
directives for worksharing of loops, clauses for data transfers between CPU and GPU
unified source codes for both serial and parallel execution
similar in both Fortran (!$ACC) and C (#pragma acc)

A closer look:

`!$ACC REGION`	a compute region: a code to be run on GPU
`!$ACC DO PARALLEL VECTOR SEQ`	mapping a loop onto a grid and blocks or a sequential run
`!$ACC DATA REGION`	one-way or two-way data transfers

and also
more directives: `!$ACC UPDATE HOST|DEVICE [ASYNC]`, `!$ACC WAIT`
more data clauses: `COPY, COPYIN, COPYOUT, LOCAL, DEVICE RESIDENT, MIRROR, REFLECTED`
more runtime functions (`acc_set_device, acc_set_device_num`),
environment variables (`ACC_DEVICE=nvidia|host, ACC_DEVICE_NUM=nr, ACC_NOTIFY`)
and compiler options

More in...
PGI Accelerator Quick Reference Card
http://www.pgroup.com/lit/literature/pgi_accel_qsg.pdf
PGI Compiler User's Guide and Reference Manual
http://www.pgroup.com/resources/docs.htm

3. PGI Accelerator directives

Idioms

a compute region with two-way data transfer

```
!$ACC REGION COPY (a)
!$ACC DO
do i=1,n ; a(i)=a(i)+z ; enddo
!$ACC END REGION
```

two compute regions nested in one data region with specified direction of data transfer

```
!$ACC DATA REGION COPYIN (a) COPYOUT (b,c)
!$ACC REGION
!$ACC DO
do i=1,n ; b(i)=a(i) ; enddo
!$ACC END REGION
...
!$ACC REGION
!$ACC DO
do j=1,n ; c(i)=a(i) ; enddo
!$ACC END REGION
!$ACC END DATA REGION
```

3. Example with PGI directives: array update, a(:)=a(:)+z

a CPU version ... pgfortran -fast t-cpu.f90

```
MODULE m
IMPLICIT NONE
INTEGER,PARAMETER :: DP=4,NMAX=4096*256

CONTAINS

SUBROUTINE Update(a,z)
REAL(DP) :: a(:)
REAL(DP) :: z
INTEGER  :: j


do j=1,size(a)
  a(j)=a(j)+z
enddo

END SUBROUTINE

END MODULE

PROGRAM Test_CPU
USE m
IMPLICIT NONE
REAL(DP) :: a(NMAX),z

a=0.
z=1.
call Update(a,z)
print *,a(1),sum(a)

END PROGRAM
```

a GPU version ... pgfortran -fast -ta=nvidia -Minfo=accel t-pgi-dirs.f90

```
MODULE m
IMPLICIT NONE
INTEGER,PARAMETER :: DP=4,NMAX=4096*256

CONTAINS

SUBROUTINE Update(a,z)
REAL(DP) :: a(:)
REAL(DP) :: z
INTEGER  :: j
!$ACC REGION
!$ACC DO
do j=1,size(a)
  a(j)=a(j)+z
enddo
!$ACC END REGION
END SUBROUTINE

END MODULE

PROGRAM Test_GPU
USE m
IMPLICIT NONE
REAL(DP) :: a(NMAX),z

a=0.
z=1.
call Update(a,z)
print *,a(1),sum(a)

END PROGRAM
```

3. OpenACC directives

In short: the brand-new specification (November 2011) of accelerator directives
proposed by NVIDIA, PGI, Cray and CAPS
PGI directives (also Cray efforts) served as a prototype
OpenACC directives are similar to original PGI directives but not the same (e.g., reductions)
new terms: gang (~ grid), worker (~ warp), vector (~ thread), sequential runs possible
PGI (beta) implementation emerged in ver. 12.3, most features expected in ver. 12.5

A closer look:

`!$ACC PARALLEL`	parallel execution of the surrounded code
`!$ACC KERNELS`	worksharing of loops, similar to ACC REGION
`!$ACC LOOP GANG WORKER VECTOR SEQ`	similar to ACC DO
`!$ACC DATA COPYIN () COPYOUT () COPY ()`	similar to ACC DATA REGION transfers

More in...
OpenACC Home, Specification, Quick Reference Guide and FAQ
    http://www.openacc-standard.org
    http://www.openacc.org/sites/default/files/OpenACC_API_QuickRefGuide.pdf
PGI Compiler User's Guide and Reference Manual, road map for OpenACC support
    http://www.pgroup.com/resources/docs.htm
    http://www.pgroup.com/resources/accel.htm#accrm
Wolfe M., The PGI Accelerator Compilers with OpenACC, 2012
    http://www.pgroup.com/lit/articles/insider/v4n1a1.htm

3. Example with OpenACC directives: array update, a(:)=a(:)+z

a CPU version ... pgfortran -fast t-cpu.f90

```
MODULE m
IMPLICIT NONE
INTEGER,PARAMETER :: DP=4,NMAX=4096*256

CONTAINS

SUBROUTINE Update(a,z)
REAL(DP) :: a(:)
REAL(DP) :: z
INTEGER  :: j


do j=1,size(a)
  a(j)=a(j)+z
enddo

END SUBROUTINE

END MODULE

PROGRAM Test_CPU
USE m
IMPLICIT NONE
REAL(DP) :: a(NMAX),z

a=0.
z=1.
call Update(a,z)
print *,a(1),sum(a)

END PROGRAM
```

a GPU version ... pgfortran -fast -acc -Minfo=accel t-pgi-oacc.f90

```
MODULE m
IMPLICIT NONE
INTEGER,PARAMETER :: DP=4,NMAX=4096*256

CONTAINS

SUBROUTINE Update(a,z)
REAL(DP) :: a(:)
REAL(DP) :: z
INTEGER  :: j
!$ACC KERNEL
!$ACC LOOP
do j=1,size(a)
  a(j)=a(j)+z
enddo
!$ACC END KERNELS
END SUBROUTINE

END MODULE

PROGRAM Test_GPU
USE m
IMPLICIT NONE
REAL(DP) :: a(NMAX),z

a=0.
z=1.
call Update(a,z)
print *,a(1),sum(a)

END PROGRAM
```

4. Initial value problems with PGI CUDA Fortran

Lecture notes (100+ slides)
ODEs (Runge-Kutta for Lorenz system)
Laplace's and Poisson eqution
the heat equation in 2D and 3D by Jacobi iterations with block and tiling approaches,
by the ADI method and by the method of lines (MOL)
the wave equation

PGI CUDA Fortran source codes
compute-bound and memory-bound kernels
single/double precision performance
kernel execution timeouts
performance impacts of various grid and block sizes
block and tiling approaches in 2D and 3D
eploiting on-chip shared memory
linking Fortran compilers (pgfortran, ifort, gfortran, g95) with CUBLAS and CULA libraries

GPU libraries for linear algebra
CUBLAS by NVIDIA (BLAS Levels 1, 2 and 3)
CULA Dense for dense linear algebra (LAPACK)
CULA Sparse for Krylov methods (CG, BiCG and GMRES with Jacobi and ILU0 preconditioners)
MAGMA for dense linear algebra (LAPACK)

5. PGI compilers on the Cascade system

cascade.msi.umn.edu

Dell CPU: 8 dual X5675 nodes each with 2x6 cores
NVIDIA GPU: 32 M2070 nodes each with 448 cores (cc 2.0)

Login: ssh cascade.msi.umn.edu

Look for PGI compilers: module avail pgi (as of 05/2012: pgi 9.04, 10.9, 11.7, 12.3)

Connect to GPUs (for 1 node with 12 CPU cores and 4 GPUs):

```
qsub -I -lwalltime=1:00:00,nodes=1:ppn=12:gpus=4
```

Load PGI module, show information, transfer and compile source files:

```
module load pgi/12.3
pgaccelinfo
pgfortran -V ; pgfortran -help
pgfortran -fast -Mcuda -acc -Minfo files.f90
```

Read more: https://www.msi.umn.edu/hpc/cascade

5. Benchmark: tested GPUs

CC	name	CUDA cores	dmem	SP Gflops	DP Gflops	power
Selected CC 2.0 products						
2.0	Tesla M2070	14 x 32 = 448	6 GB	1030	515	
2.0	GeForce GTX 590	2 x 16 x 32 = 1024	2 x 1.5 GB	1244 x 2	622 x 2 (?)	365 W
Selected CC 3.0 products						
3.0	GeForce GTX 680	8 x 192 = 1536	2 GB	(?)	1/24 of SP	195 W
Selected CC 1.3 products						
1.3	GeForce GTX 260	27 x 8 = 216	1 GB	912 (715)	1/8 of SP	182+ W
This CC 2.1 notebook						
2.1	GeForce GT 425M	2 x 48 = 96	1 GB	215	1/12 of SP	
Selected CPU						
Intel Core i7 950/3.06 GHz (Nehalem)		4		49	1/2 of SP	130 W

(theoretical) Gflops = processor_clock_in_MHz * CUDA_cores * operations_per_clock / 1000
operations_per_clock = 2 (FMA) on CC 1.x, 2 on CC 2.x, possibly 3 (FMA+SF) on Tesla, 4 on Intel Nehalem
FMA = fused multiply-add, fma(x,y,z)=x*y+z, SF = special function

5. Benchmark: matrix multiplication

Hardware

CPU: Intel Core i7 950/3.06 GHz
GTX 260: GeForce 1.3, GTX 590: top GeForce 2.0, GTX 425M: this notebook 2.1, GTX 680: 1st GeForce 3.0
M2070: Tesla 2.0 (Cascade)

Algorithm

multiplication of dense matrices 5120 x 5120 (268 Gflop, 0.3/0.6 GB), single/double precision (SP/DP)
(MKL) Intel MKL
(PGI-CUF) CUDA Fortran, (PGI-DIRS) PGI directives, (PGI-OACC) OpenACC directives
(CUBLAS) CUBLAS 4.1/4.2

Run times (sec.) on CPU and GTX 590

	MKL	PGI-CUF	PGI-DIRS	PGI-OACC	CUBLAS
SP	11.1	3.6	3.9	3.0	0.7
DP	22.2	5.1	5.1	5.6	2.3

Run times (sec.) on GPUs by CUBLAS

	GTX 260	GTX 590 (½)	GTX 425M	GTX 680	M2070
SP	1.3	0.7	3.5	0.5	1.1
DP	4.3	2.3	16.7	2.7	1.7

Run times = data transfers + computations; initialization time minimalized by pgcudainit.

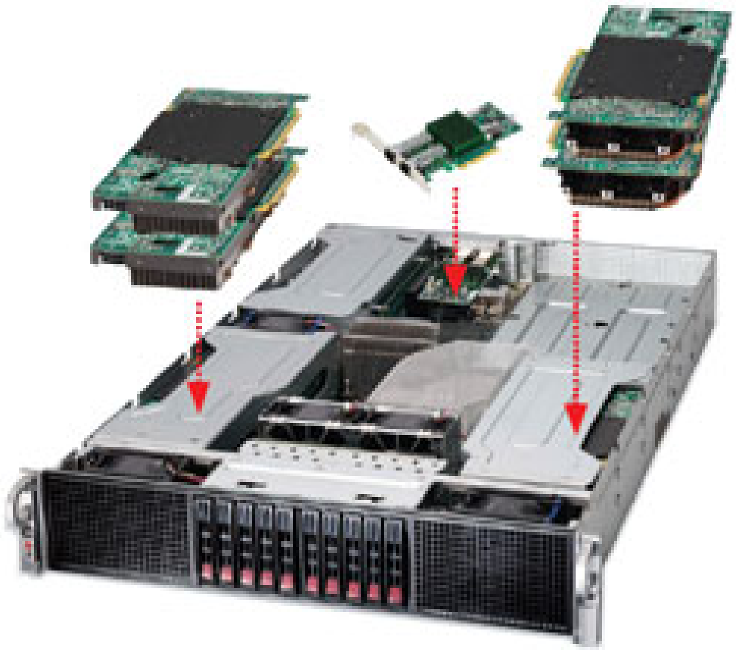
Story of implementation of matmul in CUBLAS: Volkov and Demmel, 2008 (Best Student Paper Award)
f90 source codes: mm-mkl, mm-pgi-cuf, mm-pgi-dirs, mm-pgi-oacc, mm-cublas (try on the Cascade!)

Conclusions

Since 2010, the PGI compiler suite supports GPU computing

- by CUDA extensions to Fortran 2003 and C99 compilers
- by OpenMP-like accelerator directives for both Fortran and C compilers
- a new OpenACC specification of accelerator directives partially implemented
- the PGI suite available on the Cascade system with 32 Tesla GPUs

The Portland Group



Links and references

NVIDIA hardware

http://developer.nvidia.com/cuda-gpus
http://www.nvidia.com/tesla

NVIDIA GPU Computing Documentation (e.g., NVIDIA CUDA C Programming Guide)

http://developer.nvidia.com/nvidia-gpu-computing-documentation

PGI resources

Articles, PGInsider newsletters, White papers and specifications, Technical papers and presentations
http://www.pgroup.com/resources/articles.htm

OpenAcc

http://www.openacc-standard.org

CULA Tools (libraries for dense and sparse linear algebra)

http://www.culatools.com

MAGMA (a library for dense linear algebra)

http://icl.cs.utk.edu/magma

Matrix multiplication on GPUs

Volkov V., Demmel J. W., Benchmarking GPUs to tune dense linear algebra, 2008
http://www.cs.berkeley.edu/~volkov

Minnesota Supercomputing Institute: the Cascade system

https://www.msi.umn.edu/hpc/cascade

This presentation, lecture notes (Solving PDEs with PGI CUDA Fortran), Fortran source codes

http://geo.mff.cuni.cz/~lh