Solving PDEs with PGI CUDA Fortran
Part 1: Introduction to NVIDIA hardware and CUDA architecture

Outline

Multiprocessors and compute capability. Floating-point arithmetic, Gflops. CUDA programming model: threads, blocks and grids, warps and kernels. Memory hierarchy. Compute-capability limits. memory coalescing. A kernel source code.

Accelerators

coprocessors for offloading compute-intensive processes

GPUs (graphics processing units) – coprocessors specialized to accelerate graphics
but evolved recently to serve for general-purpose (GP) GPU computing
– massively parallel: collect many (hundreds) processors (cores)
– appropriate algorithms may get speedups of 10x-100x, but redesign of applications is necessary

NVIDIA

CUDA – the most popular GP GPU parallel programming model today
– from notebooks and personal desktops to high performance computing (HPC)
– 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

nvcc and CUDA API (Application Programming Interface)
– C/C++ based proprietary compiler and library provided by NVIDIA
– many third-party tools on top of nvcc...

Portland Group Inc. (PGI): a Fortran compiler with CUDA extensions
– a high-level programming model that interoperates with highly-tuned low-level kernels: CUDA Fortran
– directive-based programming: PGI Accelerator (a software model for coding hardware accelerators)
– access to optimized GPU libraries

NVIDIA GPU generations 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): double-precision (DP)
models: GeForce GTX 295, Quadro FX 5800, Tesla C1060, S1070

Fermi/GF100/GT300 (since 2010): compute capability 2.0, 2.1
features (2.0): 32 cores/multiprocessor, faster DP, hardware cache
models: GeForce GTX 580, Quadro 6000, Tesla C2050, S2070

Product families: GeForce for games and PC graphics, Quadro for professional graphics, Tesla for HPC

A first view of NVIDIA hardware – Fermi (CC 2.0)

a device: a) 1–16 streaming multiprocessors (SMs)
b) device memory of about GB size, L2 cache of 768 KB

a multiprocessor: a) 32 thread processors (CUDA cores) for integer and SP/DP real, 4 SP special function units (SFUs)
b) registers: 128 KB, L1 cache + shared memory: 64 KB, constant cache: 8 KB, texture cache: 6–8 KB
c) 2 instruction (warp) schedulers

one device: up to 16 SMs, i.e., 16 x 32 = 512 CUDA cores
one graphics card: up to 2 devices
one motherboard: up to 2 graphics cards
a rack solution: 4 devices per module

Comparison with multicore-CPU terminology

NVIDIA terms		parallel-computing terms
a device	~	a multicore processor with each core able to run independent to another (MIMD parallelism)
a multiprocessor	~	a (vector) core with the ability to switch among several (vector) instruction streams

		(interleaved multithreading)
CUDA cores	~	scalar units executing concurrently a vector instruction stream
		(SIMD parallelism)

see Wolfe (2010) about Intel Knights Ferry versus Fermi

Other compute capabilities

CC 1.3
a multiprocessor: 8 CUDA cores for integer and SP real, 1 DP real unit, 2 SP SFUs, 1 instruction scheduler
64 KB registers/SM, 16 KB smem/SM, 8 KB cmem cache/SM, 6–8 KB texture cache
according to NVIDIA documentation no L1 & L2 cache, but there is some (e.g., Volkov 2008)
devices with up to 30 SMs, i.e., 30 x 8 = 240 CUDA cores/device
CC 2.1
a multiprocessor: 48 CUDA cores, 4 DP instructions per clock cycle, 8 SP SFUs, 2 instruction schedulers
on-chip memory and L2 cache same as CC 2.0

Gflops by NVIDIA GPUs and Intel CPUs

Giga=10^9, flops = flop/s = floating-point operations/s
(theoretical) Gflops = processor_clock_in_MHz * CUDA_cores * operations_per_clock / 1000
operations_per_clock = 2 (FMA) on CC 1.x, 2 (FMA) on CC 2.x, possibly 3 (FMA+SF) on Tesla, 4 on Intel Nehalem

Top CC 2.0 products (June 2011)

CC	name	CUDA cores	dmem	SP Gflops	DP Gflops	power
2.0	Tesla S2050	4 x 14 x 32 = 1792	12 GB	4122	2061	900 W
2.0	Tesla M2090	16 x 32 = 512	6 GB	1331	665	? W
2.0	Tesla C2070	14 x 32 = 448	6 GB	1030	515	247 W
2.0	GeForce GTX 590	2 x 16 x 32 = 1024	3 GB	2488	1244	365 W
2.0	GeForce GTX 580	16 x 32 = 512	1.5 GB	1581	790	244 W
Top CC 1.3 products						
1.3	Tesla S1070	4 x 30 x 8 = 960	16 GB	2765	346	700 W
1.3	Tesla C1060	30 x 8 = 240	4 GB	622	78	188 W
1.3	GeForce GTX 295	2 x 30 x 8 = 480	1.8 GB	1788	224	289 W
GPUs and CPUs for ~ USD 300						
2.0	GeForce GTX 470	14 x 32 = 448	1.3 GB	1089	1/2 of SP	215 W
1.3	GeForce GTX 260	27 x 8 = 216	0.9 GB	912 (715)	1/8 of SP	182+ W
Intel Core i7 950 (Nehalem Bloomfield)		4 cores		49	1/2 of SP	130 W
This notebook						
2.1	GeForce GT 425M	2 x 48 = 96	1.0 GB	215	1/12 of SP	
Intel Core i7 740QM (Nehalem mobile)		4 cores		28	1/2 of SP	45 W

Throughput of native arithmetic instructions per multiprocessor (operations per clock cycle per multiprocessor)

	integer +	integer *,FMA	SP +,*,FMA	DP +,*,FMA	SP SF (frcp, log2f, exp2f, sinf, cosf)
1.x	8	multiple	8	1	2
2.0	32	16	32	16	4
2.1	48	16	48	4 (slow!)	8

FMA = fused multiply-add, fma(x,y,z)=x*y+z, SF = special function
SP = (4B) single-precision real, DP = (8B) double-precision real
(NVIDIA CUDA C Programming Guide, Chap. 5)

Gflops/W

GeForce CC 2.0	4-7 Gflops/W
GeForce CC 1.3	3-6 Gflops/W
Intel i7 950	0.4 Gflops/W

CUDA software architecture

CUDA (Compute Unified Device Architecture): a general purpose parallel computing architecture
- hardware: multiprocessor, cores, memory
- software: a programming model
  - C/C++ compiler nvcc

CUDA API (Application Programming Interface) library

more CUDA tools by NVIDIA:

CUDA Toolkit with nvcc, CUDA debugger, Visual Profiler
GPU-accelerated numerical libraries: CUBLAS, CUSPARSE, CUFFT, CURAND
Computing SDK (Software Development Kit) code samples

more languages by third parties:

OpenCL (Khronos), Brook (Stanford University) – based on C language
Microsoft DirectCompute – a part of DirectX
PGI compiler suite (Portland Group) – PGI CUDA Fortran, PGI CUDA C/C++, PGI Accelerator
Jacket (AccelerEyes) – platform for Matlab
and many others

CUDA programming model

in hardware: a device with multiprocessors (MIMD parallelism)
a multiprocessor with CUDA cores (SIMD parallelism)
in software: a grid of blocks
a block of threads

– blocks correspond to multiprocessors, a grid to a device
– a thread is executed by a CUDA core
– all threads of a block are executed by CUDA cores of a single multiprocessor
– threads of different blocks can be executed by different multiprocessors, each independent of another ("MIMD")

More about grids and blocks

– grids and blocks are effectivelly 1D, 2D or 3D indexed arrays of threads
– blocks are limited in size (~1024 threads), to fit well into 32 cores of a multiprocessor
– a grid size is effectivelly unlimited (~ 2^48 ~ 10^14 blocks)
– an optimal block size should be chosen carefully in order to reach a high multiprocessor occupancy
(i.e., a number of threads resident in a multiprocessor)
– a grid size is chosen to meet a problem size with a given block size, block size * grid size = problem size
– a device with more multiprocessors can process a large grid faster

Moreover, there are warps:
– groups (vectors) of 32 consecutive threads of a block that are executed in parallel in hardware
("SIMD", in CUDA rather SIMT: single-instruction multiple-threads)
– warps in a block are executed concurrently, but one at a time ("interleaved multithreading"),
they are switched by warp schedulers
– threads in a warp are free to branch and execute independently, but a performance of a warp
would be reduced (divergent warps)
– threads in a warp can benefit from access patterns to device memory that can be merged into one transaction
(memory coalescing), e.g., addressing consecutive elements of a properly aligned array

Kernel

– a procedure launched from the host and executed on the device
– a source code is written as for a single thread and executed by all threads
– the kernel executes asynchronously, i.e., the host process continues concurrently
– the host and the device are synchronized implicitly at the point of host-device memory transfer,
or explicitly by a synchronization routine
– some devices are capable of execution concurrent with memory transfers
– the total number of threads, i.e., grid and block sizes, is set dynamically at the time of kernel launch

GPU memory hierarchy – Fermi (CC 2.0)

On device...
– device memory (dmem) used for – global memory: public data, shared by threads
– local memory (lmem): private data, local in threads, did not fit into registers
– constant memory: data initialized by the host, read-only in the device
– texture memory: data initialized by the host, read-only in the device
– L2 cache for faster access to device memory, shared by all multiprocessors
On each multiprocessor („on-chip“)...

– registers: local data, also used internally by the compiler
– L1 cache: for faster access to device memory, shared by all CUDA cores in a multiprocessor
– shared memory (smem), shared by all CUDA cores in a multiprocessor („software-managed cache”)
– available configurations: 16 KB L1 cache + 48 KB smem or 48 KB L1 cache + 16 KB smem
– 8 KB constant cache: for faster reading from 64 KB constant memory (cmem) residing in dmem
– 6–8 KB texture cache: for faster reading from texture memory residing in dmem, optimized for 2D arrays

About the latency (Volkov 2008)...
– registers: no (read) latency
– smem (i.e., L1 cache): units or tens of clock cycles
– dmem: hundreds of clock cycles

and the memory bandwidth...
– transfers in dmem: from tens to above 100 GB/s
– host-dmem transfers: 6 GB/s (PCI Express 2.0) or less

On the host side...
– host memory can be allocated as pinned (page-locked):
pinned host-dmem transfers are faster by tens of % up to two times
the page-locked memory may not be available

CUDA compute capability limits

(NVIDIA CUDA C Programming Guide, App. F, also CUDA_Occupancy_Calculator.xls)

Grid and block related limits

1.x	max block dimensions: 512-512-64, but total size: 512 threads/block
	max grid dimensions: 65535-65535-1 (max 2D grids)
2.x	max block dimensions: 1024-1024-64, but total size: 1024 threads/block
	max grid dimensions: 65535-65535-65535 (3D grids)
1.0,1.1	max 24 resident warps/SM, i.e., max 768 threads/SM
1.2,1.3	max 32 resident warps/SM, i.e., max 1024 threads/SM
2.0,2.1	max 48 resident warps/SM, i.e., max 1536 threads/SM
all	max 8 resident blocks/SM
	warp size: 32 threads/warp

Memory related limits

	registers	lmem	smem	cmem	cmem cache	texture cache
1.0,1.1	32 KB/SM	16 KB/thread	16 KB/SM	64 KB/device	8 KB/SM	6-8 KB/2 SMs
1.2,1.3	64 KB/SM	16 KB/thread	16 KB/SM	64 KB/device	8 KB/SM	6-8 KB/3 SMs
2.0,2.1	128 KB/SM	512 KB/thread	16-48 KB/SM	64 KB/device	8 KB/SM	6-8 KB/SM

CUDA on GeForce

a GeForce GPU is usually attached to a display and serves the graphical user interface of an operating system
the GUI is stalled during a kernel run, the display is updated between the kernel runs
there is a runtime limit for a single kernel on a GPU with a display attached:

Linux:	~ 8 s
Microsoft Windows XP:	~ 5 s
Microsoft Windows Vista, Windows 7:	~ 2 s

after that, the process calling the kernel is cancelled or the OS crash occurs
Linux: a window manager can be stopped (Ubuntu: service gdm stop), the system can then be accessed remotely and there is no timeout
Windows Vista and Windows 7 can disable or extend the limit by via registry editing or merging registry entries by the .reg scripts:
to disable Timeout Detection and Recovery (TDR)...

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GraphicsDrivers]
"TdrLevel"=dword:00000000
```

to extend the 2-s limit to 60 s...

```
Windows Registry Editor Version 5.00
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\GraphicsDrivers]
"TdrDelay"=dword:00000060
```

see CUDA_Toolkit_Release_Notes.txt or http://www.microsoft.com/whdc/device/display/wddm_timeout.mspx
(Timeout Detection and Recovery of GPUs through WDDM)

Finally, a first example: addition of a 1D array and a scalar, a(:)=a(:)+z

a CPU version
pgfortran -fast t1c.f90

```
MODULE mConst

INTEGER,PARAMETER :: DP=4,NMAX=4096*256

END MODULE

MODULE mProc
USE mConst
IMPLICIT NONE

CONTAINS

SUBROUTINE Assign(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 Template_1_CPU
USE mConst
USE mProc
IMPLICIT NONE
REAL(DP) :: a(NMAX),z


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

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

END PROGRAM
```

a GPU version
pgfortran -fast -Mcuda t1g.f90

```
MODULE mConst
USE cudafor
INTEGER,PARAMETER :: DP=4,NG=4096,NB=256,NMAX=NG*NB
TYPE(dim3),PARAMETER :: grid=dim3(NG,1,1),block=dim3(NB,1,1)
END MODULE

MODULE mProc
USE mConst
IMPLICIT NONE

CONTAINS

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

END SUBROUTINE

END MODULE

PROGRAM Template_1_GPU
USE mConst
USE mProc
IMPLICIT NONE
REAL(DP) :: a(NMAX),z
REAL(DP),DEVICE :: ad(NMAX)

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

END PROGRAM
```

Differences between CPU and GPU versions:
– initialization: cudafor module, grid and block shape and size
– a kernel: global attribute, attributes of arguments, outer loops replaced by thread indexing
– a kernel call: allocation of device data, host-device data transfers, executable configuration

Examples in CUDA Fortran SDK folder

bandwidthTest
goal: speed of CPU-GPU and GPU-GPU data transfers

Links and references

NVIDIA hardware

http://www.nvidia.com/tesla	etc.
http://en.wikipedia.org/wiki/Nvidia_Tesla	etc.

NVIDIA GPU Computing Documentation
 NVIDIA CUDA C Programming Guide (esp., Chap. 4 & 5 & App. A & F)
 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

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

Wolfe M., Compilers and More: Knights Ferry versus Fermi, 2010
 http://www.hpcwire.com/hpcwire/2010-08-05/compilers_and_more_knights_ferry_versus_fermi.html

Solving PDEs with PGI CUDA Fortran
Part 2: Introduction to PGI CUDA Fortran

Outline

Why Fortran, why CUDA Fortran. Compilers for NVIDIA GPUs. Hierarchy of CUDA Fortran, CUDA C and CUDA Runtime API. Kernel and device subroutines. GPU memory specification and allocation. Host-device data transfers. Launching kernels. Porting source codes: to-do list. Compiler switches. Source-code examples.

Why Fortran

– a well-established programming language for scientific and engineering applications
– supports high performance computing and parallelization (OpenMP, MPI, ...)
– ready-to-link optimized numerical libraries (LAPACK, NAG, IMSL, ...)
– hides some technicalities ("pointers and asterisks are not for everybody")
– standardized interoperability with C
– and of course, http://www.pbm.com/~lindahl/real.programmers.html

Why PGI CUDA Fortran and PGI Accelerator

– CUDA Fortran: a small set of extensions to Fortran that supports and is built upon the CUDA computing architecture
– PGI Accelerator: directives similar to OpenMP style to define accelerated regions and corresponding data
– a Fortran programmer can keep living in the Fortran world:
  device data can be declared and allocated by Fortran specification and allocation statements
  host-device data transfer can be done by Fortran assignment statements (including array syntax)
  kernels can be written and launched using extended Fortran syntax
– CUDA Fortran is higher-level programming model relative to CUDA C
– PGI Accelerator is higher-level programming model relative to CUDA Fortran

Compilers for NVIDIA GPUs

nvcc: a free C/C++ proprietary compiler by NVIDIA, included in the CUDA Toolkit
  a command-line tool on top of a standard C/C++ compiler
  provides calls to (higher-level) CUDA Runtime API and (lower-level) CUDA Driver API
  June 2011: release version 4.0; previous versions 3.2, 3.1, 2.3

PGI Workstation: a commercial compiler suite by PGI (the Portland Group)
  Fortran 95/2003 and C/C++ compilers
  for Linux, Windows, MacOS, 32 and 64 bits, with OpenMP and MPI support
  the command-line interface, for Windows: the Microsoft Visual Studio Express environment
  GPU support: CUDA Fortran, CUDA C/C++, PGI Accelerator directives
  selected parts of CUDA Toolkit in the suite: nvopencc, CUDA Runtime API, CUBLAS, CUFFT
  June 2011: release version 11.6 with CUDA Toolkits 3.2 and 4.0

a working set: NVIDIA graphics driver (with support of CUDA Toolkit used by PGI)
  PGI Workstation or PGI Server (Linux, Windows, MacOS) or PGI Visual Fortran (Windows)
  optional: NVIDIA CUDA Toolkit (requires gcc on Linux, MS Visual C++ Express on Windows)

CUDA Fortran mission

writing kernel subroutines and device procedures
```
ATTRIBUTES(KERNEL) SUBROUTINE MyKernel(arguments)
```
declaring and allocating data in the device or on-chip memory
```
INTEGER,ALLOCATABLE,DEVICE :: ad(:) ! dmem
REAL :: b                           ! registers or lmem
REAL,CONSTANT :: pi                 ! cmem
COMPLEX,SHARED :: c(nmax)           ! smem
```
transferring data between host and device
```
ad=a ; ... ; a=ad
```
launching kernels
```
call MyKernel<<<gridsize,blocksize>>>(arguments)
```
calling CUDA Runtime API routines
```
istatus=cudaThreadSynchronize()
```
accessing definitions of CUDA types and interfaces
```
use cudafor
```

Kernel subroutine and device procedures

Kernel subroutine
– launched by the host for execution on GPU
– specified by ATTRIBUTES(GLOBAL)
– written for a single thread, executed by each thread (the total number of threads is assigned by the kernel call)
– typically updates an array passed as an argument, often one array element by one thread

Supported datatypes
– INTEGER(1,2,4,8), REAL(4,8), COMPLEX(4,8), LOGICAL(1,2,4,8), CHARACTER(1) and derived types

Supported statements
– assignment statements, including the array language
– conditional statements and constructs IF and SELECT CASE
– loops DO with index variable, DO WHILE and unbounded DO, along with CYCLE and EXIT statements
– statements CONTINUE, GOTO, CALL and RETURN

Supported Fortran and CUDA intrinsics
Fortran: abs, aimag, aint, ..., min, max, ..., acos, asin, atan, ..., all, any, count, maxloc, maxval, sum, ...
CUDA: fma_rn, fma_rz, fma_ru, fma_rd, fmaf_rn, ... (many others)

Constraints
– cannot contain STOP and PAUSE
– cannot contain input/output commands (PRINT *,scalar possible since ver. 11.6)
– cannot contain ALLOCATABLE or POINTER data
– cannot be RECURSIVE, PURE or ELEMENTAL
– cannot contain procedures, cannot be contained in a procedure (can appear in a module)

Arguments
– arrays must be ready in device memory already, scalars can be in host memory
– actual arguments in device memory are passed by reference (arrays always, scalar optionally)
    and are shared by all threads – they are in global memory (residing in device memory),
    corresponding dummy arguments have the DEVICE attribute by default
– actual arguments in host memory (scalars only) are passed by value
    and are private to each thread – they are in registers or in local memory (residing in device memory),
    corresponding dummies must have the VALUE attribute

Device procedures
– subroutines and functions that can be called from a kernel or another device procedure (not from a host procedure)
– specified by ATTRIBUTES(DEVICE)

Thread indexing
– a unique thread index that can (must) be found from built-in variables...

a) threadidx	for the index of a thread in a block (1 is the lowest)
b) blockidx	for the index of a block in a grid (also one-based)
c) blockdim	for the size and shape of a block
d) griddim	for the size and shape of a grid

– the variables are structures of type dim3: type dim3 ; integer x,y,z ; end type
    i.e., they allow for 1D (via x component), 2D (x,y) and 3D (x,y,z) block and grid shapes

Correspondence between array indexes and thread and block indexes can be done in many ways,
    but the threadidx%x should always correspond to consecutive array elements (for memory coalescing)
– 1D array, 1D block, 1D grid: i=threadidx%x+blockdim%x*(blockidx%x-1)
– 1D array, 2D block, 1D grid: i=threadidx%x+blockdim%x*((threadidx%y-1)+blockdim%y*(blockidx%x-1))
– 2D array, 1D block, 1D grid: i=threadidx%x, j=blockidx%x etc.
– Fortran-matrices have the column-major order ("1st index is changing fastest"),
    i.e., threadidx%x should correspond to the array index for the first dimension

Synchronization
– threads running in parallel occasionally need synchronization at a certain point (a barrier)
– a barrier for all threads of a block is a call to a CUDA routine: call syncthreads() and variants
– a barrier for all threads of a grid is the end of the kernel: end subroutine

GPU memory specification and allocation

Device memory (dmem)

– resides on the device, accessible by both the host and the device
– bandwidth to the host side slower (via PCI Express slot, ~< 6 GB/s), to multiprocessors faster (~< 100 GB/s)
– used primarily for global memory (public data shared by all threads),
    also for local memory (private data local in each thread if registers are exhausted and spilled),
    also for constant memory (read-only public data, cached on multiprocessors),
    also for texture memory (similar, but not accessible by CUDA Fortran)
– provides a two-way bridge between the host and the device:
    a host procedure declares, allocates and (optionally) initializes data in dmem,
    the dmem variables are passed as kernel arguments by reference,
    if updated in the kernel, the host procedure can copy the dmem data back to host memory
– variables in dmem are declared with the DEVICE attribute:
    if in a host procedure, the variable can appear only in allocations and deallocations, in assignment
        statements (as the source and/or the destination) and as an actual or dummy argument
    if in a kernel as a dummy argument, the variable is public and shared by all threads
        (the DEVICE attribute of dummy arguments is implicit for actual arguments from dmem)
    if in a kernel as a local variable, the variable is private in each thread
– device arrays in a host procedure can be both static and allocatable, local arrays in kernels can be static only
– if threads of a warp access dmem arrays by consecutive elements, they will all be served simultaneously
    because of the device memory coalescing
– the ordering of multidimensional arrays in linear memory must be taken into account:
    the column-major order used by Fortran (as opposite to C) implies that the thread index threadidx%x
    should be used as the array index for the first dimension
– fully fledged dmem coalescing also requires correct alignment of accessed array elements but failing to do so
    will result in two memory accesses at worst

Registers and local memory (lmem)

– registers reside in fast on-chip memory, lmem in slower device memory, but it can be stored in L1 or L2 cache
– registers readable with no latency, dmem latency is hundreds of clock cycles, cache latency in-between
– used for private data local in each thread, not accessible by the host
– variables in registers and lmem can appear in kernels and device procedures, not in host procedures
– variables in registers and lmem are declared without any CUDA attribute
– kernel dummy arguments with the VALUE attribute are stored into registers or lmem

Constant memory (cmem)

– resides in device memory (64 KB) and can be cached on each multiprocessor (8 KB)
– must be initialized in a host procedure and is read-only in a kernel and device procedures
– variables in cmem are declared with the CONSTANT attribute which is illegal in host procedures, thus,
    cmem data should be declared in a module with a kernel and made visible to a host procedure
– cmem broadcasting: when more threads of a warp read the same word from cmem cache, they will all
    be served simultaneously

Shared memory (smem)

– resides in fast on-chip memory of 16 KB (CC 1.x/2.x) or 48 KB (CC 2.x) on each multiprocessor
– used for data shared by all threads in a block (but not among different blocks)
– variables in smem are declared with the SHARED attribute in a kernel or a device procedure
    and must be initialized by threads
– the amount of smem required by a block poses a limit on a number of resident blocks in a multiprocessor,
    e.g., real(8) shared array of 1024 elements occupies 8 KB and, with 48 KB smem per multiprocessor,
    max. 6 blocks can be resident on a multiprocessor (but 8 blocks is the upper limit anyway)
– smem allows fast access as is (an order of magnitude faster than uncached access to dmem)
– moreover, when threads of a warp access consecutive 4B words in smem, they will all be served simultaneously
    because of the ability of smem to access smem banks simultaneously
    (consecutive words are assigned to consecutive smem banks)
– smem broadcasting: when more threads of a warp read the same word from smem, they will all be served
    simultaneously
– a smem bank conflict occurs when two or more threads of a warp access a different word in the same smem bank

Host-device data transfer

– simple assignment statements in host procedures

– statements provided for:	host-to-device transfer,	`ad=a`
	device-to-host transfer,	`a=ad`
	device-to-device transfer,	`bd=ad`

– for arrays, one contiguous block transfer shoud be preferred
– similarly, host-to-constant-memory transfer can be issued
– on systems with integrated host and device memory, the mapped page-locked memory should be used,
 data transfer would then be superfluous
– CUDA memory management routines are also provided

Launching kernels

– a kernel call statement is expected to set the number of threads that will execute the kernel by assigning
 the execution configuration, i.e., actual grid and block sizes
– grid and block sizes are either scalar integers or dim3 structures,

```
grid=dim3(1024,1024,1), block=256
```

– extended form of the CALL statement:

```
CALL Kernel<<<grid,size>>>(arguments)
```

– the chevrons can optionally specify amount of dynamically assigned shared memory (a scalar integer bytes)
 and the stream identification (0 or a scalar integer returned by the cudaStreamCreate function)

```
<<<grid,size,bytes,stream>>>
```

– grid and block sizes have to satisfy compute-capability limits, dynamically assigned smem must be available

Porting source codes from CPU Fortran to CUDA Fortran: To-do list

1. extract parallelizable portions of the code (most often with one or more loops)
 into subroutines contained in a module
2. edit the kernel:
 set the global attribute
 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

Compiler switches

pgfortran -help [option]	or pgf77, pgf90, pgf95
pgfortran -V	version information
pgfortran file.f90	no optimization
pgfortran -fast file.f90	common local optimization set, see pgfortran -help -fast
pgfortran -fastsse file.f90	more local optimizations on 32bit systems (equivalent to -fast on 64bit systems)
pgfortran -fast -Mipa=fast file.f90	global optimization
pgfortran -g file.f90	debugging information
pgfortran -Mcuda file.f90	enable CUDA Fortran

Mcuda suboptions: -Mcuda=cc11,cc13,cc20,3.1,3.2,4.0,emu,keepgpu,ptxinfo etc.

cc11, cc13, cc20	specific compute capability (default: all; cc21 not yet available)
3.1, 3.2, 4.0	specific CUDA Toolkit compatibility (default in ver. 11.6: 3.2, in 11.5: 3.1)
emu	emulation mode
keepgpu	keeping kernel CUDA C source files
ptxinfo	messages from ptxas about register and lmem/smem/cmem usage (ptxas = PTX-language assembler; PTX = Parallel Thread Execution)

pgaccelinfo utility
CUDA Fortran source coudes can have .cuf extension, then the -Mcuda option is default

CUDA Fortran source-code examples

Example 1 (template): addition of 1D array and a scalar
a(:)=a(:)+z
goals: setting the execution configuration – the grid and block sizes
passing arguments to a kernel – arrays by reference, scalars by value
correspondence of 1D arrays and thread indexing

Example 2: addition of 2D arrays by a device function
a(:,:)=a(:,:)+b(:,:)
goals: correspondence of 2D arrays and thread indexing
device procedures

Example 3: accessing 2D arrays in nested loops
a(:,:)=a(:,:)+b(:,:) once again
goals: efficient access to 2D arrays with column-major order on CPU and GPU
device memory coalescing

Example 4: avoiding divergent warps
each thread summing the harmonic series $\sum_{n=1}^{N} 1/n$
goals: execution time for various grid and block sizes
execution time for various amounts of diverging execution paths in a warp

Example 5: using shared memory (the 3-point moving average filter)
the moving average = a finite-impulse response filter that creates a series of averages of the original data set
for $n=0,1,2,\dots$: $a_0^{n+1}=(2a_0^n+a_1^n)/3$, for $j=1,\dots,J$: $a_j^{n+1}=(a_{j-1}^n+a_j^n+a_{j+1}^n)/3$, $a_J^{n+1}=(a_J^n+a_{J+1}^n)/3$
goals: transfer of device memory data to shared memory
synchronization of threads in a block
and...

A final example: Mandelbrot set

wiki: a set of points, whose boundary generates a two-dimensional fractal shape
a set M of complex numbers c for which the $lim_{n\to\infty}|z_n|$ of the sequence $z_{n+1}=z_n^2+c$, $z_0=0$, remains bounded
c is not in M, if $|z_n|>2$ for any n
a source-code snippet:

```
complex cc,z ;
z=0.; do n=1,nmax ; z=z*z+cc ; if (abs(z)>2.) record_cc_and_exit ; enddo
```

note: abs(z)>2. may be rather slow, real(z)**2+imag(z)**2>4. is expected to evaluate faster
vizualization of Mandelbrot-set approximations:
for each c, the highest n, if any, for which abs(z_n)<=2, is recorded
all c corresponding to a fixed n form a set M_n, the n-th approximation of M
all M_n are vizualized, each with a different color

Links and references

PGI resources
CUDA Fortran Programming Guide and References, Release 2011
PGI Compiler User's Guide, Release 2011 (chapter Using an Accelerator)
PGI Compiler Reference Manual, Release 2011 (chapter PGI Accelerator Compilers Reference)
Articles, PGInsider newsletters, White papers and specifications, Technical papers and presentations
http://www.pgroup.com/resources/articles.htm

NVIDIA GPU Computing Documentation
NVIDIA CUDA C Programming Guide (esp., Chap. 5: Performance guidelines)
NVIDIA CUDA C Best Practices Guide
NVIDIA Tuning CUDA applications for Fermi
http://developer.nvidia.com/nvidia-gpu-computing-documentation

Source-code examples
CUDA Fortran SDK: C:\Program Files\PGI\win64\2011\cuda\CUDA Fortran SDK

Wolfe M., CUDA Fortran: The next level, PGInsider, 2010
http://www.pgroup.com/lit/articles/insider/v2n3a1.htm

Solving PDEs with PGI CUDA Fortran
Part 3: Linear algebra. Laplace's equation

Outline

Compute- and memory-bound kernels. Matrix multiplication. Optimized libraries for linear algebra. Direct and iterative methods for linear algebraic equations. Laplace's and Poisson's equations in 1D. Direct solution and Jacobi and Gauss-Seidel iterations.

Compute-bound and memory(-bandwidth)-bound kernels

Definitions

number of floating-point operations	F [flop]	for SP (single) or DP (double precision)
floating-point operations per second	dF [flop/s]	
number of transferred bytes or words	B [byte] or W [word]	W=B/4 for SP, W=B/8 for DP
memory bandwidth per second	dB [bytes/s] or dW [words/s]	
float:byte ratio	F/B or dF/dB	[flop/bytes]
float:word ratio	F/W or dF/dW	[flop/word]
relation	F/W = 4 F/B	

Theoretical hardware limits	dF SP [Gflop/s]	dB [GB/s]	dF/dB [flop/byte]
Tesla C2070	1030	144	7.2
GeForce GTX 470	1089	134	8.1
GeForce GTX 260	715	112	6.4
GeForce GT 425M	215	25.6	8.4

Theoretical limits in basic linear-algebra algorithms

	F	W	F/W	F/B
dot-product of vectors of n elements	2n	2n+n = 3n	2/3	2.7
matrix-vector product	n . 2n	$n^2+n+n^2 \sim 2n^2$	1	4
matrix-matrix product	$n^2 . 2n$	$2n^2+n^2 = 3n^2$	2n/3	2.7 n

i.e., only the matrix multiplication provides the flop:byte ratio large enough for GPUs
and therefore can be compute-bound, while BLAS 1+2 algorithms are memory-bound

Matrix multiplication

see PGI CUDA Fortran User Guide, chapter Examples for the source code
see Volkov's paper for a story of developing matmul on GPUs

Optimized libraries for linear algebra

BLAS library (Basic Linear Algebra Subroutines)
for summs, scaling, dot products, matrix multiplication etc.
Levels 1 (vector-vector), 2 (matrix-vector), 3 (matrix-matrix)
in single precision (prefix s), double precision (d), complex SP (c), complex DP (z)

e.g.,	L1: saxpy	alpha*x(1:n)+y(1:n)
	L3: dgemm	alpha*A(1:m,1:k)*B(1:k,1:n)+beta*C(1:,1:n)

a port for GPUs by NVIDIA: the CUBLAS library

LAPACK library (Linear Algebra PACKage)
for solving linear algebraic equations by direct methods, also for eigenvalue problems
includes – BLAS Levels 1, 2, 3
– direct solvers of linear algebraic systems with general, band-diagonal matrices, symmetric positively definite matrices etc.
– algorithms: LU, QR and Cholesky factorization, singular value decomposition (SVD) etc.
– dense, banded and other matrices
a port available in commercial optimized general-purpose libraries: MKL, IMSL, NAG
variants for sparse matrices and for parallel computing
for GPUs: packages CULA tools (for fee), MAGMA (for free)

Running pgfortran with CUBLAS

CUBLAS – a part of CUDA Toolkit, i.e. recent versions: 3.1, 3.2, 4.0
packed in the directory tree of PGI
works with dmem arrays, can be linked with gfortran/g95/ifort (examples in App. B of CUBLAS User Guide),
 but with PGI it's simple (examples in CUDA Fortran SDK in PGI tree and now)
example with GEMM: TestCUBLAS.f90
 goals: interoperability of Fortran and C, Fortran interface to overloaded subroutines, random numbers

Running pgfortran with CULA tools

CULA tools – Basic version free for SP, Premium version with DP and more routines
 – recent versions R11 (for CUDA 3.2), R12 (for CUDA 4.0)
 – Fortran-relevant interfaces: Fortran for host-mem arrays, Device for dmem arrays
performance graphs
examples with GEMM: TestCULA.f90
 goals: interface to CULA functions, CULA initialization, inquiries and shutdown

Direct and iterative methods for linear algebraic equations

Linear algebraic systems

$$Ax = b, \qquad a_{ij}x_j = b_i, \quad i,j = 1,\dots,n$$

Direct methods

solve for a unique solution (if it exists) in ~ n^3 operations (for dense matrices)
i.e., for n=10^3: number of operations ~ 10^9, for n=10^4: ~ 10^{12} operations, etc.
no additional information necessary

LU factorization for general (dense) matrices
 a) decomposition A = L . U , where L is a lower triangular matrix, U an upper triangular matrix
 b) solving to algebraic equations L . y = b for vector y
 c) solving to algebraic equations U . x = y for vector x
 variants for band diagonal (banded) matrices available

Cholesky decomposition for symmetric positive definite matrices
 possible to decompose into the form A = L . L^T

Iterative methods

solve for approximate solutions iteratively

$$x^{n+1} = Hx^n + g, \qquad n = 0, \dots$$

i.e., an initial approximation x^0 must be provided
 and matrix-vector multiplication is performed in each iteration
the only way when the matrix A is large
typical requirements for success:
 the iteration matrix is sparse
 the iterations converge rather fast
 e.g., for ~ n operations for matrix-vector multiplications and ~ n or ~ const number of iterations,
 the total number of operations may be linear (~ n) or quadratic (~ n^2) function of n
methods: Jacobi, Gauss-Seidel, successive overrelaxation (SOR), conjugate gradient method (CGM),
 multigrid method (MG) etc.

Laplace's and Poisson's equations in 1D

Equation and boundary conditions

the second-order differential equation for a real function u(x) of one real variable x

$$\Delta u(x) \equiv \frac{d^2u(x)}{dx^2} \equiv u''(x) = f(x)$$

the right-hand side

- Laplace: $f(x) = 0$
- Poisson: $f(x) \neq 0$

boundary conditions

- Dirichlet: $u(x_0) = u_0, \quad u(x_{\max}) = u_{\max}$
- Neumann: $u'(x_0) = u'_0$ or $u'(x_{\max}) = u'_{\max}$ (not simultaneously at both ends)

i.e., the boundary value problem (BVP) for the elliptic differential equation

Features of the solution to the Laplace's equation (i.e., harmonic functions)

- the maximum principle: extremes of u(x) always at the boundary
- the mean value theorem: integral over a ball is proportional to the value in the center of the ball

in 1D: $u(x_S) = \left(\int_{x_0}^{x_{\max}} u(x)\,dx \right) / (x_{\max} - x_0)$

Analytical solutions

Laplace's equation:	$u(x) = bx + c$
Poisson's equation with constant f(x) = a:	$u(x) = ax^2/2 + bx + c$
Poisson's equation with arbitrary f(x):	$u(x) = \iint f(x) + bx + c$

where b and c can be obtained from the two boundary conditions

Discretization

the equidistant grid

$$x_j, \quad j = 0, \dots, J+1$$
$$x_j = x_0 + j\,dx, \quad dx = (x_{J+1} - x_0)/(J+1)$$
$$u_j \equiv u(x_j)$$

the centered 2nd-order finite-difference scheme for the 2nd derivative (FD2)

$$u''(x_j) \approx \frac{u_{j-1} - 2u_j + u_{j+1}}{dx^2}$$

the left- and right-hand 1st-order finite-difference schemes for the 1st derivative, needed for the Neumann conditions

$$u'(x_0) \approx \frac{u_1 - u_0}{dx}, \qquad u'(x_{J+1}) \approx \frac{u_{J+1} - u_J}{dx}$$

Discretized system of linear algebraic equations

– for the Dirichlet boundary conditions at both ends

$$\begin{aligned} 2u_1 - u_2 &= -f_1\,dx^2 + u_0 \\ -u_{j-1} + 2u_j - u_{j+1} &= -f_j\,dx^2, \qquad j = 1+1, \dots, J-1 \\ -u_{J-1} + 2u_J &= -f_J\,dx^2 + u_{J+1} \end{aligned}$$

i.e., the matrix of the system takes the tridiagonal (moreover, symmetric and positive definite) form

$$\begin{pmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \cdot & \cdot & \cdot & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \cdot \\ u_{J-1} \\ u_J \end{pmatrix} = \begin{pmatrix} -f_1\,dx^2 + u_0 \\ -f_2\,dx^2 \\ \cdot \\ -f_{J-1}\,dx^2 \\ -f_J\,dx^2 + u_{J+1} \end{pmatrix}$$

– for the Neumann boundary condition at one or the other end, the first or the last equation is different

$$u_1 - u_2 = -f_1\,dx^2 - u'_0\,dx$$

or

$$-u_{J-1} + u_J = -f_J\,dx^2 + u'_{J+1}\,dx$$

Direct solution

Linear algebraic equations with tridiagonal matrix (see Numerical Recipes chapter 2.3)

- a) a loop to eliminate of subdiagonal elements
- b) a loop to eliminate superdiagonal elements ("backsubstitution")

Example of direct solution to 1D Laplace's and Poisson's equations

– based on the serial routine tridag (Numerical Recipes Chapter 2.4)

Iterative solution

Decomposition A = L + D + U with L lower triangular, D diagonal and U upper triangular matrix

Then, (L + D + U) . x = b

can be rewritten into the form suitable for iterations,

$$x = D^{-1} (b - (L + U) . x)$$

or, for each row,

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j - \sum_{j=i+1}^{J} A_{ij} x_j \right), \quad i = 1, \ldots, J$$

Jacobi iterations

the iteration index n is appended to both L . x and U . x terms

$$x^{n+1} = D^{-1} (b - (L+U) x^n)$$

$$x_i^{n+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^n - \sum_{j=i+1}^{J} A_{ij} x_j^n \right), \quad i = 1, \ldots, J$$

Gauss-Seidel iterations

the iteration index n is appended to U . x term only, the n+1 goes to L . x

$$x^{n+1} = (L+D)^{-1} (b - U x^n)$$

$$x_i^{n+1} = \frac{1}{a_{ii}} \left(b_i - \sum_{j=1}^{i-1} A_{ij} x_j^{n+1} - \sum_{j=i+1}^{J} A_{ij} x_j^n \right), \quad i = 1, \ldots, J$$

Features:

– updates in Jacobi iterations have to be stored into new memory positions and can therefore be performed in parallel

– Gauss-Seidel iterations update the existing memory positions and are supposed to be performed serially

– Gauss-Seidel is proved to be slightly more accurate for some matrices

Examples of Jacobi and Gauss-Seidel-like iterations for 1D Laplace's equation

Links and references

Libraries

CUBLAS Library User Guide
http://developer.nvidia.com/nvidia-gpu-computing-documentation

CULA Programmers Guide and Reference Manual
http://www.culatools.com/features/performance

MAGMA, Matrix Algebra on GPU and Multicore Architectures
http://icl.cs.utk.edu/magma/

Calling CUBLAS from CUDA Fortran, 2010
http://cudamusing.blogspot.com/

Humphrey J., Spagnoli K., Using the CULA GPU-enabled LAPACK Library with CUDA Fortran, PGInsider, 2010
http://www.pgroup.com/lit/articles/insider/v2n3a5.htm

Tomov S. et al., Using MAGMA with PGI Fortan, PGInsider, 2010
http://www.pgroup.com/lit/articles/insider/v2n4a4.htm

Toepfer C., Using GPU-enabled Math Libraries with PGI Fortran, PGInsider, 2011
http://www.pgroup.com/lit/articles/insider/v3n1a5.htm

Matrix multiplication

Volkov V., Demmel J. W., Benchmarking GPUs to Tune Dense Linear Algebra, 2008
http://www.cs.berkeley.edu/~volkov/

Volkov V., Demmel J. W., LU, QR and Cholesky factorizations using vector capabilities of GPUs, 2008

Nath R. et al., An Improved MAGMA GEMM for Fermi GPUs, 2010
http://icl.cs.utk.edu/projectsfiles/magma/pubs/fermi_gemm.pdf

Numerical methods

Press W. H. et al., Numerical Recipes in Fortran 77: The Art of Scientific Computing, Second Edition, Cambridge, 1992
Chapter 2.3: LU decomposition and its applications
Chapter 2.4: Tridiagonal and band diagonal systems of equations
Chapter 19.0: Partial differential equations – Introduction
http://www.nr.com, PDF available at http://www.nrbook.com/a/bookfpdf.php

Solving PDEs with PGI CUDA Fortran
Part 4: Initial value problems for ordinary differential equations

Outline
ODEs and initial conditions. Explicit and implicit Euler methods. Runge-Kutta methods. Multistep Adams' predictor-corrector and Gear's BDF methods. Example: Lorenz attractor.

Ordinary differential equations and initial conditions
1 ordinary differential equation for 1 unknown function y(x) of 1 variable x

$$dy(x)/dx \equiv y'(x) = f(x, y(x))$$

For a unique solution, the initial condition is required

$$y(x_0) = y_0$$

A set of M ordinary differential equations for M unknown functions $y_m(x)$ of 1 variable x

$$dy_m(x)/dx = f_m(x, y_1(x), \dots, y_M(x)), \quad m = 1, \dots, M$$

or $\quad dY(x)/dx \equiv Y'(x) = F(x, Y(x))$

for vector Y of unknown functions y_m and vector F of right-hand-side functions f_m

For a unique solution, M initial conditions are required

$$y_m(x_0) = y_{m0} \quad \text{or} \quad Y(x_0) = Y_0$$

These are initial value problems (IVPs) for ordinary differential equations (ODEs).
Higher-order ODEs can be rewritten into a set of 1st-order ODEs (see, e.g., Numerical Recipes).

Discretization
x-grid and stepsize $\quad x_n, \quad h_n = x_{n+1} - x_n, \quad n = 0, 1, \dots$

numerical solution $\quad y_n \approx y(x_n)$

Numerical methods below are, for simplicity, formulated for 1 ODE and the constant stepsize h.
However, they all also work for y replaced by Y and h replaced by h_n.

Explicit Euler method
the left-hand 1st-order finite-difference scheme for the 1st derivative

$$y'(x_n) \approx (y(x_{n+1}) - y(x_n))/h$$

after substitution into the ODE, we get the approximate Euler method

$$y_{n+1} = y_n + hf(x_n, y_n)$$

it is an explicit formula as all terms on the right-hand side are known

accuracy: 1st-order method (corresponds to the truncated Taylor expansion with the 0th and 1st term only)

stability: consider a linear problem with constant coefficients

$y'(x) = -cy, \quad y(0) = 1, \quad c > 0$ (solution: $y(x) = e^{-cx}$)

Euler method: $y_{n+1} = y_n + h(-cy_n) = (1 - ch)y_n$

but for $h > 2/c : |y_{n+1}| > |y_n|, \text{ or } |y_n| \to \infty$; thus, there is a stepsize limit due to stability

pros:	a simple explicit formula	
cons:	low accuracy	=> higher-order explicit methods
	low stability	=> implicit methods

Implicit Euler method
the right-hand 1st-order finite-difference scheme for the 1st derivative

$$y'(x_{n+1}) \approx (y(x_{n+1}) - y(x_n))/h$$

after substitution into the ODE, we get another Euler method

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$$

it is an implicit formula as there are references to unknown y_{n+1} on the right-hand side

again, it is only 1-st order accurate

as above, consider the problem

$y'(x) = -cy, \quad y(0) = 1, \quad c > 0$ (solution: $y(x) = e^{-cx}$)

the implicit formula: $\quad y_{n+1} = y_n + h(-cy_{n+1}) = y_n/(1 + ch)$

thus, implicit Euler method is stable for any (positive) h, it has an infinite region of absolute stability

Semi-implicit Euler method for $Y' = F(Y)$
solving implicit Euler method by linearization of F(Y), similarly as in the Newton method for root finding

$$Y_{n+1} = Y_n + hF(Y_{n+1}) \approx Y_n + h\left[F(Y_n) + (\partial F/\partial Y)_{Y_n} \cdot (Y_{n+1} - Y_n)\right]$$

$$Y_{n+1} = Y_n + h[I - h(\partial F/\partial Y)]^{-1} \cdot F(Y_n)$$

i.e., in each step, MxM (Jacobian) matrix assembly and inversion is required

Runge-Kutta methods

– more accurate, higher-order methods for integrating ODEs
– approximate the Taylor expansion by averaging the appropriately chosen dy/dx along y(x) between x_n and x_{n+1}
– RK1: the explicit Euler method, the simplest RK method

$$y_{n+1} = y_n + hk_1$$
$$k_1 = f(x_n, y_n)$$

for f(x) independent of y, it is equivalent to the rectangle quadrature rule
– RK2: the 2nd-order RK method (midpoint method)

$$y_{n+1} = y_n + hk_2$$
$$k_1 = f(x_n, y_n), \quad k_2 = f(x_n + h/2, y_n + hk_1/2)$$

for f(x) independent of y, it is equivalent to the trapezoidal quadrature rule
– RK4: the most popular, 4th-order RK method

$$y_{n+1} = y_n + h(k_1 + 2k_2 + 2k_3 + k_4)/6$$
$$k_1 = f(x_n, y_n)$$
$$k_2 = f(x_n + h/2, y_n + hk_1/2)$$
$$k_3 = f(x_n + h/2, y_n + hk_2/2)$$
$$k_4 = f(x_n + h, y_n + hk_3)$$

for f(x) independent of y, it is equivalent to the Simpson's quadrature rule
– a general p-step (explicit) RK method

$$y_{n+1} = y_n + h\sum_{i=1}^{p} c_i k_i$$
$$k_1 = f(x_n, y_n)$$
$$k_i = f(x_n + \alpha_i h, y_n + h\sum_{j=1}^{i-1} \beta_{ij} k_j), \quad i = 2, \ldots, p$$

– stepsize control (different h in each step): guess a step size,
or make a few runs with step doubling,
or use RK methods with adaptive stepsize control (e.g., Numerical Recipes, Chapter 16.2)
– pros: higher accuracy, better stability (not as much as in implicit variants) than explicit Euler
– implicit RK methods available (more stable than the explicit methods, but not infinitely stable)

Multistep methods

A general linear multistep method

$$y_{n+1} = \sum_{i=1}^{p} a_i y_{n+1-i} + h\sum_{i=0}^{p} b_i f_{n+1-i} \qquad \text{with } a_p \text{ or } b_p \text{ nonzero}$$

the methods are explicit and p-step for b_0=0 and implicit and (p+1)-step otherwise

Adams' family of multistep methods
– based on polynomial approximation of f(x,y(x)) between x_{n+1-p} and x_n (x_{n+1} for implicit methods)
and analytical integration of the approximating polynomial between x_n and x_{n+1}

$$y_{n+1} = y_n + \int_{x_n}^{x_{n+1}} N_p(x)\,dx = y_n + h\sum_{i=0}^{p} b_i f_{n+1-i}$$

– the order (i.e., the coincidence with the truncated Taylor expansion) is p for explicit and p+1 for implicit methods
– the explicit p=1 method is the explicit Euler, the implicit p=0 method is the implicit Euler

Coefficients a_i and b_i of Adams' methods

explicit (Adams-Bashforth) methods

all p: a_1 = 1, other a_i = 0

	i:	0	1	2	3	4
p=0:	b_i	–				
p=1:	b_i	0	1			
p=2:	$2b_i$	0	3	−1		
p=3:	$12b_i$	0	23	−16	5	
p=4:	$24b_i$	0	55	−59	37	−9

e.g., explicit p=2: $y_{n+1} = y_n + h(3f_n - f_{n-1})/2$

implicit (Adams-Moulton) methods

all p: a_1 = 1, other a_i = 0

	i:	0	1	2	3	4
p=0:	b_i	1				
p=1:	$2b_i$	1	1			
p=2:	$12b_i$	5	8	−1		
p=3:	$24b_i$	9	19	−5	1	
p=4:	$720b_i$	251	646	−264	106	−19

implicit p=1: $y_{n+1} = y_n + h(f_{n+1} + f_n)/2$

The predictor-corrector algorithm: conventional application of Adams' methods
step P (predictor): applies an explicit Adams' formula of a given order, ynew = y_{n+1}[explicit](x_{n+1})
step E (evaluation): updates f_{n+1} = f(x_{n+1},ynew)
step C (corrector): applies an implicit Adams' formula of the same order, ynew = y_{n+1}[implicit](x_{n+1})
steps E and C can be repeated: variants PEC, PECE, P(EC)^{2}E
i.e., a predictor extrapolates f into x_{n+1}, a corrector makes use of this value for polynomial interpolation

– initialization of multistep methods by their lower-order relatives or by RK methods
– the predictor-corrector algorithm is essentially explicit, its stability is therefore worse than that of the corrector
– adaptive stepsize control is laborious

Backward differentiation formulas (BDFs, Gear's method)
– based on polynomial approximation of y(x) between x_{n+1-p} and x_{n+1} and analytical differentiation of the approximating polynomial at x_{n+1}
$y_{n+1} = \sum_{i=1}^{p} a_i y_{n+1-i} + hb_0 f_{n+1}$
– implicit p-step methods of the order p; for p=1: implicit Euler method
– BDFs combined with the Newton method are known to have excellent stability (for $p \le 6$)
– inevitable for stiff problems with two or more very different scales of the variable x on which the unknowns y are changing (stability conditions require to accommodate to the fastest scale, i.e., with small stepsize, while the process under study usually develops on the slowest scale, i.e., too many small steps would be necessary with explicit methods)

Coefficients a_i and b_i of Gear's methods

	i:	1	2	3	4	5	6	i:	0	> 0
p=1:	a_i	1						b_i	1	0
p=2:	$3a_i$	4	−1					$3b_i$	2	0
p=3:	$11a_i$	18	−9	2				$11b_i$	6	0
p=4:	$25a_i$	48	−36	16	−3			$25b_i$	12	0
p=5:	$137a_i$	300	−300	200	−75	12		$137b_i$	60	0
p=6:	$147a_i$	360	−450	400	−225	72	−10	$147b_i$	60	0

e.g., p=2: $y_{n+1} = (4y_n - y_{n-1})/3 + 2f_{n+1}/3$

On the crossroads

Euler methods are extremely simple but inaccurate, too; the implicit variant is necessary when stability matters, i.e., when larger stepsize is required than a stability condition allows.
Runge-Kutta explicit methods are simple and fast enough both to code and to run, as each step requires just evaluation of an explicit formula, and for many problems, they are accurate enough. However, they are explicit and stepsize is limited.
Predictor-corrector implementation, including stepsize adaptivity, is rather an artwork, but it was done and can be reused from available packages. Still, stability is limited.
Backward differentiation formulas, as a multistep method, share many features with predictor-corrector methods, however, for their excellent stability, they are inevitable for stiff problems.

And an example: Lorenz attractor

– a problem of the 2D convection in the atmosphere, mathematically simplified as much as possible
– a fully deterministic system with chaotic behavior
– a simplified problem: 3 ODEs for temporal evolution of 3 variables (coefficients of eigenvalue expansions of the stream function and temperature anomalies) in the 3D (phase) space

$$\begin{aligned} dA/d\tau &= P(B-A) \\ dB/d\tau &= rA - B - AC \\ dC/d\tau &= -bC + AB \end{aligned}$$

where A is a stream-function coefficient, B and C coefficients of temperature anomalies, P the Prandtl number, $r=Ra/Ra_{CR}$ with Ra the Rayleigh number, $b = 4/(1+(2/\lambda)^2)$ corresponds to the size of a convection roll and τ is nondimensionalized time
– parameters used by Lorenz (1963): P=10, r=28 and b=8/3, a sufficient time interval: 0..20
– temporal solutions roll around two fixed points (the strange attractors) along a lemniscate-shaped trajectory (like ∞)
– physically: the boundary layer of a convection cell grows, at some point it becomes unstable, convection resumes, either as a clockwise or counterclockwise roll: chaotic behavior in a deterministic system
– popular vizualization: A-B-C phase portraits

Source codes
– CPU: an arbitrary A-B-C point undergoes NT Runge-Kutta time steps, they are recorded and plotted
– CPU: NX x NY x NZ points, spread within a 3D cube, undergo NT time steps, each independent of others, only final positions of all points are recorded and plotted

– GPU: the previous case with one kernel
– GPU: the previous case, now with a smaller kernel called repeatedly
Goals: a massively parallel compute-bound kernel, SP/DP execution times, avoiding kernel execution timeout, stability limits of explicit schemes

Links and references

Numerical methods

Ascher U. M. and Petzold L. R., Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations, SIAM, 1998

Press W. H. et al., Numerical Recipes in Fortran 77: The Art of Scientific Computing, Second Edition, Cambridge, 1992
Chapter 16.1: Runge-Kutta method
Chapter 16.2: Adaptive stepsize control for Runge-Kutta
Chapter 16.6: Stiff sets of equations
http://www.nr.com, PDFs available at http://www.nrbook.com/a/bookfpdf.php

Lorenz attractor

Schubert G. et al., Mantle Convection in the Earth and Planets, Cambridge, 2001, p. 332–337
http://ebookee.org/Mantle-Convection-in-the-Earth-and-Planets_661884.html

Solving PDEs with PGI CUDA Fortran
Part 5: Explicit methods for evolutionary partial differential equations

Outline

Heat equation in one, two and three dimensions. Discretization stencils. Block and tiling implementations. Method of lines.

Heat equation

temporal evolution (physically, diffusion) of heat (temperature) in a domain
a partial differential equation (1-st order in time t, 2-nd order in spatial variables X) for a function u(t, X)
1D (one-dimensional) case: X = x, 2D case: X = x,y, 3D case: X = x,y,z

General form: $\partial_t u(t,X) = \Delta u(t,X)$
in 3D: $\partial_t u(t,x,y,z) = (\partial_x^2 + \partial_y^2 + \partial_z^2) u(t,x,y,z)$
Initial condition: $u(t_0,X) = u_0(X)$
Boundary conditions: $u(t,X_B) = u_B(t,X_B)$ on the boundary
i.e., the initial value problem (IVP) for the parabolic partial differential equation

Discretization grids and schemes

the equidistant grid on a rectangular domain, constant time steps

$$\begin{aligned}
t_n &= t_0 + n\,dt, & dt &= (t_N - t_0)/N \\
x_j &= x_0 + j\,dx, & dx &= (x_J - x_0)/J \\
y_k &= z_0 + k\,dy, & dy &= (y_K - y_0)/K \\
z_l &= z_0 + l\,dz, & dz &= (z_L - z_0)/L \\
u^n_{jkl} &\approx u(t_n, x_j, y_k, z_l)
\end{aligned}$$

moreover, $J = K = L, \quad dx = dy = dz$

Explicit FTCS scheme (forward-in-time, centered-in-space)
FD1 for time: $\partial_t u^n_{jkl} \approx (u^{n+1}_{jkl} - u^n_{jkl})/dt$ (cf. Euler method for ODEs)
FD2 for space:
$$\partial_x^2 u^n_{jkl} \approx (u^n_{j-1,k,l} - 2u^n_{jkl} + u^n_{j+1,k,l})/dx^2$$
$$\partial_y^2 u^n_{jkl} \approx (u^n_{j,k-1,l} - 2u^n_{jkl} + u^n_{j,k+1,l})/dy^2$$
$$\partial_z^2 u^n_{jkl} \approx (u^n_{j,k,l-1} - 2u^n_{jkl} + u^n_{j,k,l+1})/dz^2$$
More spatial stencils:
FD4 $\partial_x^2 u^n_j \approx \frac{1}{12}(-u^n_{j-2} + 16u^n_{j-1} - 30u^n_j + 16u^n_{j+1} - u^n_{j+2})/dx^2$
FD6 $\partial_x^2 u^n_j \approx \frac{1}{180}(2u^n_{j-3} - 27u^n_{j-2} + 270u^n_{j-1} - 490u^n_j + 270u^n_{j+1} - 27u^n_{j+2} + 2u^n_{j+3})/dx^2$

Discretized heat equation in 1D

1D heat equation $u^{n+1}_j = (1-2\beta)u^n_j + \beta\left(u^n_{j-1} + u^n_{j+1}\right), \qquad \beta = dt/dx^2$
accuracy: 1st-order in time, 2-nd order in space
stability condition: $\beta \le 1/2, \quad dt \le dx^2/2, \quad N \ge 2J^2(t_N - t_0)/(x_J - x_0)^2$

The sinus example

domain $t_0 = 0,\ 0 \le x \le 1$
initial condition $u_0(x) = \sin(\pi x)$
boundary conditions constant and consistent with the initial condition
analytical solution
$$u(t,x) = e^{-\pi^2 t}\sin(\pi x)$$
minimal number of timesteps to reach t = 1, according to the stability condition, is N = 2 J^2

Equilibrium solution of the heat equation

In the equilibrium limit, $\partial_t u = 0$, the heat equation takes form of the Laplace's equation,
i.e., long-time solutions of the heat equation converge to the solutions of the Laplace's equation.
Iterations $u_j^{n+1} = (1-2\beta)u_j^n + \beta\left(u_{j-1}^n + u_{j+1}^n\right)$ are called the Jacobi iterations, as they,
in the stability limit of $\beta = 1/2$, take form of $u_j^{n+1} = \left(u_{j-1}^n + u_{j+1}^n\right)/2$,
that we have already called the Jacobi iterations for the 1D Laplace's equation.

Discretized heat equation in 2D

2D heat equation $u_{jk}^{n+1} = (1-4\beta)u_{jk}^n + \beta\left(u_{j-1,k}^n + u_{j+1,k}^n + u_{j,k-1}^n + u_{j,k+1}^n\right), \qquad \beta = dt/dx^2$

the stability condition $\beta \le 1/4, \quad dt \le dx^2/4, \quad N \ge 4J^2(t_N - t_0)/(x_J - x_0)^2$

The 2D sinus example

domain $t_0 = 0,\ 0 \le x \le 1,\ 0 \le y \le 1$
initial condition $u_0(x,y) = \sin(\pi x)\sin(\pi y)$
boundary conditions constant and consistent with the initial condition
analytical solution

$$u(t,x,y) = e^{-2\pi^2 t}\sin(\pi x)\sin(\pi y)$$

minimal number of timesteps to reach t = 1, according to the stability condition, is N = 4 J^2

GPU implementations of Jacobi iterations in 2D

Block approach

– the spatial domain is split into rectangular blocks (not necessarily squares)
– each block of grid points (with halo or ghost points on block boundaries) is assigned to 1 CUDA block
– each thread updates one grid point
Notes:
CUDA blocksize limit of 1024 threads/block corresponds to number of grid points, i.e., max. 32x32 (32x16, 64x8, ...)
smem limit of 48 KB/multiprocessor: 4+ KB for a SP array of 32x32 grid points
more work in a kernel: merging (e.g., 4) grid points for 1 thread
higher-order spatial discretization (FD4 etc.)
keeping CUDA blocks smaller makes better multiprocessor occupancy (up to 8 blocks/multiprocessor)
allows for implementation of wildly asynchronous kernels

Tiling approach

– the spatial domain is split into rectangular strips
– each strip of grid points (with halos on strip boundaries) is assigned to 1 CUDA block
– each thread updates one line of grid points
– a 1D temporary smem array (a tile, degenerated in 2D to an abscissa) moves along these lines
together with two abscissas made from registers
Notes:
– CUDA blocksize ~ 64, 128, 256, e.g., for 1024^2 grid points and CUDA block size of 128, there is 8 CUDA blocks
– smem limit high enough
– well suited for FD4 etc.

Discretized heat equation in 3D

3D heat equation $u_{jkl}^{n+1} = (1-6\beta)u_{jkl}^n + \beta\left(u_{j-1,kl}^n + u_{j+1,kl}^n + u_{j,k-1,l}^n + u_{j,k+1,l}^n + u_{j,k,l-1}^n + u_{j,k,l+1}^n\right)$
$\beta = dt/dx^2$

the stability condition $\beta \le 1/6, \quad dt \le dx^2/6, \quad N \ge 6J^2(t_N - t_0)/(x_J - x_0)^2$

The 3D sinus example

domain $t_0 = 0,\ 0 \le x \le 1,\ 0 \le y \le 1,\ 0 \le z \le 1$
initial condition $u_0(x,y,z) = \sin(\pi x)\sin(\pi y)\sin(\pi z)$
boundary conditions constant and consistent with the initial condition
analytical solution

$$u(t,x,y,z) = e^{-3\pi^2 t}\sin(\pi x)\sin(\pi y)\sin(\pi z)$$

minimal number of timesteps to reach t = 1, according to the stability condition, is N = 6 J^2

GPU implementations of Jacobi iterations in 3D

Block approach

size 3D blocks of grid points substantially limited by the CUDA blocksize limit of 1024 threads/block (e.tg., 16x8x8)

Tiling approach

– the spatial domain is split into rectangular columns
– each column of grid points (with halos on column boundaries) is assigned to 1 CUDA block
– each thread updates one line of grid points
– a 2D temporary shared-memory array (the tile) moves along these lines together with two tiles made from registers

Method of lines (MOL)

motivation: use ODEs techniques for time integration instead of explicit Euler method in the FTCS scheme
procedure: discretization of spatial variables but not the time variable, i.e., from PDEs to ODEs,
and solving the ODEs with advanced solvers

Heat equation with Dirichlet boundary conditions

1D: $\partial_t u_j(t) = \beta\,(u_{j-1} - 2u_j + u_{j+1})\,, \quad \beta = 1/dx^2, \quad j = 1, \dots, J$

$$\partial_t \begin{pmatrix} u_1(t) \\ u_2(t) \\ \cdot \\ u_{J-1}(t) \\ u_J(t) \end{pmatrix} = \beta \begin{pmatrix} -2 & 1 & & & \\ 1 & -2 & 1 & & \\ & \cdot & \cdot & \cdot & \\ & & 1 & -2 & 1 \\ & & & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1(t) \\ u_2(t) \\ \cdot \\ u_{J-1}(t) \\ u_J(t) \end{pmatrix} + \beta \begin{pmatrix} u_0 \\ 0 \\ \cdot \\ 0 \\ u_{J+1} \end{pmatrix}, \quad \beta = 1/dx^2$$

2D: $\partial_t u_{jk}(t) = \beta\,(u_{j-1,k} + u_{j+1,k} + u_{j,k-1} + u_{j,k+1} - 4u_{jk})\,, \quad \beta = 1/dx^2, \quad j,k = 1, \dots, J$

etc.
On GPU, the Jacobi iterations are required, both block or tiling approaches are possible.
The GPU/CPU speedup is the same as the speedup for Jacobi iterations in the FTCS case but we received
the chance to converge faster than with the Euler method.
However, using implicit ODEs solvers should be considered.

Links and references

Numerical methods

Koev P., Numerical Methods for Partial Differential Equations, 2005
http://dspace.mit.edu/bitstream/handle/1721.1/56567/18-336Spring-2005/OcwWeb/Mathematics/18-336Spring-2005/CourseHome/index.htm
Press W. H. et al., Numerical Recipes in Fortran 77: The Art of Scientific Computing, Second Edition, Cambridge, 1992
Chapter 19.0: Introduction
Chapter 19.2: Diffusive initial value problems
Chapter 19.3: Initial value problems in multidimensions
Chapter 19.5: Relaxation methods for boundary value problems
http://www.nr.com, PDFs available at http://www.nrbook.com/a/bookfpdf.php
Spiegelman M., Myths and Methods in Modelling, 2000
http://www.ldeo.columbia.edu/~mspieg/mmm/

CUDA techniques

Micikevicius P., 3D finite difference computation on GPUs using CUDA, 2009
Rivera G. and Tseng Ch.-W., Tiling optimizations for 3D scientific computations, 2000
Venkatasubramanian S. and Vuduc R. W., Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems, 2009
Xu Ch. et al., Tiling for performance tuning on different models of GPUs, 2009

Solving PDEs with PGI CUDA Fortran
Part 6: More methods for more partial differential equations

Outline

Heat equation in 1D: implicit and Crank-Nicolson schemes. Heat equation in more dimensions: alternating-direction implicit method. Multigrid method. Wave equation in 1D and 2D: strings and drums.

Heat equation in 1D: more schemes

A symbol for the difference operator

$$\delta_x^2 u_j^n \equiv u_{j-1}^n - 2u_j^n + u_{j+1}^n$$

FTCS scheme with Dirichlet boundary conditions

$$u_j^{n+1} = u_j^n + \beta\delta_x^2 u_j^n, \qquad \beta = dt/dx^2$$
$$u_j^{n+1} = u_j^n + \beta\left(u_{j-1}^n - 2\beta u_j^n + u_{j+1}^n\right)$$

Features: 1st-order accurate in time, 2nd-order in space, conditionaly stable ($\beta <= 1/2$)

BTCS scheme (backward-time centered-space)
implicit formula

$$u_j^{n+1} = u_j^n + \beta\delta_x^2 u_j^{n+1}, \qquad \beta = dt/dx^2$$

$$\begin{pmatrix} 1+2\beta & -\beta & & & \\ -\beta & 1+2\beta & -\beta & & \\ & \cdot & \cdot & \cdot & \\ & & -\beta & 1+2\beta & -\beta \\ & & & -\beta & 1+2\beta \end{pmatrix} \begin{pmatrix} u_1^{n+1} \\ u_2^{n+1} \\ \cdot \\ u_{J-1}^{n+1} \\ u_J^{n+1} \end{pmatrix} = \begin{pmatrix} u_1^n + \beta u_0 \\ u_2^n \\ \cdot \\ u_{J-1}^n \\ u_J^n + \beta u_{J+1} \end{pmatrix}$$

Features: 1st-order accurate in time, 2nd-order in space, unconditionaly stable (i.e., for any dt)
Each time step requires direct solution to a linear algebraic system with tridiagonal matrix of size J x J.

Crank-Nicolson scheme (CN)
implicit formula with an average of FTCS and BTCS schemes on the right-hand side

$$u_j^{n+1} = u_j^n + \frac{\beta}{2}\left(\delta_x^2 u_j^n + \delta_x^2 u_j^{n+1}\right)$$

$$\begin{pmatrix} 1+\beta & -\beta/2 & & \\ -\beta/2 & 1+\beta & -\beta/2 & \\ & \cdot & \cdot & \cdot \\ & & -\beta/2 & 1+\beta \end{pmatrix} \begin{pmatrix} u_1^{n+1} \\ u_2^{n+1} \\ \cdot \\ u_J^{n+1} \end{pmatrix} = \begin{pmatrix} 1-\beta & \beta/2 & & \\ \beta/2 & 1-\beta & \beta/2 & \\ & \cdot & \cdot & \cdot \\ & & \beta/2 & 1-\beta \end{pmatrix} \begin{pmatrix} u_1^n \\ u_2^n \\ \cdot \\ u_J^n \end{pmatrix} + \begin{pmatrix} \beta u_0 \\ 0 \\ \cdot \\ \beta u_{J+1} \end{pmatrix}$$

Features: 2nd-order accurate in both time and space, unconditionally stable
Each time step requires direct solution to a linear algebraic system with tridiagonal matrix of size J x J.

Heat equation in 2D: FTCS, BTCS and CN schemes

Difference operators	$\delta_x^2 u_{jk}^n \equiv u_{j-1,k}^n - 2u_{jk}^n + u_{j+1,k}^n, \quad \delta_y^2 u_{jk}^n \equiv u_{j,k-1}^n - 2u_{jk}^n + u_{j,k+1}^n$
FTCS scheme	$u_{jk}^{n+1} = u_{jk}^n + \beta\left(\delta_x^2 u_{jk}^n + \delta_y^2 u_{jk}^n\right)$
BTCS scheme	$u_{jk}^{n+1} = u_{jk}^n + \beta\left(\delta_x^2 u_{jk}^{n+1} + \delta_y^2 u_{jk}^{n+1}\right)$
CN scheme	$u_{jk}^{n+1} = u_{jk}^n + \frac{\beta}{2}\left(\delta_x^2 u_{jk}^n + \delta_y^2 u_{jk}^n + \delta_x^2 u_{jk}^{n+1} + \delta_y^2 u_{jk}^{n+1}\right)$

For implicit BTCS and CN schemes, the matrix is J^2 x J^2, sparse and band diagonal (tridiagonal with fringes).
Direct solution is possible with special methods.

Heat equation in more dimensions: alternating-direction implicit (ADI) method

2D: splitting the time step into 2 substeps, each of lenght t/2

$$\begin{aligned} u_{jk}^{n+1/2} &= u_{jk}^{n} + \frac{\beta}{2}\left(\delta_x^2 u_{jk}^{n+1/2} + \delta_y^2 u_{jk}^{n}\right) \\ u_{jk}^{n+1} &= u_{jk}^{n+1/2} + \frac{\beta}{2}\left(\delta_x^2 u_{jk}^{n+1/2} + \delta_y^2 u_{jk}^{n+1}\right) \end{aligned}$$

3D: splitting the time step into 3 substeps, each of length t/3

$$\begin{aligned} u_{jkl}^{n+1/3} &= u_{jkl}^{n} + \frac{\beta}{3}\left(\delta_x^2 u_{jkl}^{n+1/3} + \delta_y^2 u_{jkl}^{n} + \delta_z^2 u_{jkl}^{n}\right) \\ u_{jkl}^{n+2/3} &= u_{jkl}^{n+1/3} + \frac{\beta}{3}\left(\delta_x^2 u_{jkl}^{n+1/3} + \delta_y^2 u_{jkl}^{n+2/3} + \delta_z^2 u_{jkl}^{n+1/3}\right) \\ u_{jkl}^{n+1} &= u_{jkl}^{n+2/3} + \frac{\beta}{3}\left(\delta_x^2 u_{jkl}^{n+2/3} + \delta_y^2 u_{jkl}^{n+2/3} + \delta_z^2 u_{jkl}^{n+1}\right) \end{aligned}$$

All substeps are implicit and each requires direct solutions to J independent linear algebraic systems with tridiagonal matrices of size J x J.

Example: ADI method for heat equation in 2D and 3D

Wave equation

a quantity travelling over the domain

a partial differential equation (2nd-order in time t, 2nd-order in spatial variables X) for a function u(t, X)

1D (one-dimensional) case: X = x, 2D case: X = x,y, 3D case: X = x,y,z

General form: $\partial_t^2 u(t,X) = c^2 \Delta u(t,X)$

in 3D: $\partial_t^2 u(t,x,y,z) = c^2(\partial_x^2 + \partial_y^2 + \partial_z^2) u(t,x,y,z)$

Initial conditions: $u(t_0,X) = u_0(X), \quad \partial_t u(t_0,X) = v_0(X)$

Boundary conditions: $u(t,X_B) = u_B(t,X_B)$ on the boundary

i.e., the initial value problem (IVP) for the hyperbolic partial differential equation

Discretized wave equation in 1D

1D wave equation

$$\partial_t^2 u(t,x) = c^2 \partial_x^2 u(t,x)$$

can be rewritten into the form of two equations of the 1st-order in time

$$\begin{aligned} (\partial_t + c\partial_x) u(t,x) &= v(t,x) \\ (\partial_t - c\partial_x) v(t,x) &= 0 \end{aligned}$$

Discretization grids

$$\begin{aligned} t_n &= t_0 + n\,dt, & dt &= (t_N - t_0)/N \\ x_j &= x_0 + j\,dx, & dx &= (x_J - x_0)/J \\ u_j^n &\approx u(t_n, x_j), & v_j^n &\approx v(t_n, x_j) \end{aligned}$$

Explicit FTBS scheme (forward-in-time, backward-in-space)

FD1 for time: $\partial_t u_j^n \approx (u_j^{n+1} - u_j^n)/dt$

FD1 for space: $\partial_x u_j^n \approx (-u_{j-1}^n + u_j^n)/dx$

Features: low accuracy, stability for $\frac{c\,dt}{dx} \le 1$ (Courant-Friedrichs-Lewy condition)

PDEs in the matrix form:

$$\partial_t \begin{pmatrix} u \\ v \end{pmatrix} = \begin{pmatrix} -c\partial_x & 1 \\ 0 & c\partial_x \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}$$

Discretized equations:

$$\begin{pmatrix} u \\ v \end{pmatrix}^{n+1} = \begin{pmatrix} I - \gamma A & \delta I \\ 0 & I + \gamma A \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}^{n}, \quad \gamma = \frac{c\,dt}{dx}, \ \delta = dt, \ A = \begin{pmatrix} 1 & 0 & 0 & \cdot \\ -1 & 1 & 0 & \cdot \\ 0 & -1 & 1 & \cdot \end{pmatrix}$$

and I is the identical matrix

Explicit FTCS scheme (forward-in-time, centered-in-space)

FD1 for time:	$\partial_t u_j^n \approx (u_j^{n+1} - u_j^n)/dt$
FD2 for space:	$\partial_x u_j^n \approx (-u_{j-1}^n + u_{j+1}^n)/(2dx)$

Features: unstable for any dt, i.e., FTCS scheme inappropriate for the wave equation

Implicit Crank-Nicolson scheme

implicit formula with an average of FTBS and BTBS schemes on the right-hand side

$$\begin{pmatrix} u \\ v \end{pmatrix}^{n+1} = \begin{pmatrix} u \\ v \end{pmatrix}^{n} + \begin{pmatrix} -\gamma A & \delta I \\ 0 & \gamma A \end{pmatrix} \left[\begin{pmatrix} u \\ v \end{pmatrix}^{n} + \begin{pmatrix} u \\ v \end{pmatrix}^{n+1} \right], \quad \gamma = \frac{c\,dt}{dx},\ \delta = dt,\ A = \begin{pmatrix} 1 & 0 & 0 & \cdot \\ -1 & 1 & 0 & \cdot \\ 0 & -1 & 1 & \cdot \end{pmatrix}$$

Features: higher accuracy, unconditional stability (i.e., for any dt)

Example: travelling waves

domain	$t \geq t_0 = 0,\ x \geq x_0 = 0$
initial condition	$u(0,x) = u_0(x), \quad v(0,x) = 0$
boundary condition	$u(t,0) = 0$
analytical solution	$u(t,x) = u_0(t - cx)$

Links and references

PDEs

Koev P., Numerical Methods for Partial Differential Equations, 2005
http://dspace.mit.edu/bitstream/handle/1721.1/56567/18-336Spring-2005/OcwWeb/Mathematics/18-336Spring-2005 /CourseHome/index.htm

Lehtinen J., Time-domain numerical solution of the wave equation, 2003
http://www.cs.unm.edu/~williams/cs530/wave_eqn.pdf

Piché R., Partial Differential Equations, 2010
http://math.tut.fi/~piche/pde/index.html

Press W. H. et al., Numerical Recipes in Fortran 77: The Art of Scientific Computing, Second Edition, Cambridge, 1992
Chapter 19.2: Diffusive initial value problems
Chapter 19.3: Initial value problems in multidimensions
Chapter 19.6: Multigrid methods for boundary value problems
http://www.nr.com, PDFs available at http://www.nrbook.com/a/bookfpdf.php

Spiegelman M., Myths and Methods in Modelling, 2000
http://www.ldeo.columbia.edu/~mspieg/mmm/

Wave equation on GPU

Michéa D. and Komatitsch D., Accelerating a three-dimensional finite-difference wave propagation code using GPU graphics cards, 2010