Accelerating finite differences for solving a barotropic ocean model on the GPU

Universiteit Utrecht

Folkert Bleichrodt
Utrecht University

A thesis submitted for the degree of
Master of Science

February 2011

Contents

1 Introduction to CUDA **7**

1.1 Hardware layout and thread organization 7

1.1.1 Thread model 8

1.1.2 Memory model 10

1.2 Coalesced memory access 12

1.3 Bank conflicts 13

1.4 Warp divergence 14

1.5 Performance benchmarking 15

2 The model **18**

2.1 Barotropic vorticity equation 18

2.2 Discretization 20

2.3 Poisson solver 22

2.3.1 FFT-based sine transform 24

2.4 Implementation details of the Poisson solver 26

2.4.1 Real FFT 27

2.5 Implementation details of the spatial finite difference approximation . . . 31

3 Numerical results **32**

3.1 Accuracy 35

4 Conclusions and prospects **38**

Bibliography **39**

A Hardware specifications **42**

A.1 GPU specifications 42

A.2 CPU specifications 43

B Tables **44**

B.1 Time per iteration 44

B.2 Speedup 45

Introduction

The computer and mobile devices industry today is one of the most flourishing markets in the world. It always has been since the introduction of the personal computer in the late eighties. According to Moore's Law the number of transistors of computers doubles every two years. For the past twenty years this has largely been correct. Of course, this effect can only be sustained with a driving force behind it: a thriving consumer electronics market.

While the scientific community embraces the use of High Performance Computing and has access to the fastest computers in the world, it is a relatively small user group. Therefore the decisions and the directions of hardware companies are still largely based on consumer revenues. Many scientists therefore rely on large clusters of CPUs connected by a fast network. Distributed memory systems make out most of the TOP500 [1] of super computers. Nevertheless, CPUs are not the only hardware components that are being heavily developed.

The video gaming industry has driven the development of specialized graphical hardware, the Graphics Processing Unit (GPU). Its main task is to render realistic images for video games. To keep up with the increasing complexity of games, the GPU has turned into a highly parallel device. With a large amount of processors it allows fine-grained parallelism for parallel rendering of pixels and vertices.

While the GPU has a lot of potential as a parallel device, it was close to useless for scientists. Apart from accelerated visualizations, a GPU was not suitable for general purpose computing. Some early attempts have been made [5], but the development efforts are huge, since all mathematical computations have to be translated in graphics operations.

In 2006 NVIDIA provided a toolkit which extends the C programming language with a minimal set of paradigms for parallelism which allows to use GPUs for general purpose

computing. This extension is called the *Compute Unified Device Architecture* or CUDA. With CUDA it is now possible to create parallel programs in C for the GPU, similar to openMP or openMPI programs.
The introduction of GPUs in science has proven to be successful. The TOP500 features more and more clusters extended with several GPUs per node. On NVIDIA's website applications of CUDA are being showcased and numerous scientific papers are listed showing moderate to impressive improvements in performance over a CPU.

In oceanography, the GPU still does not play a major role in numerical simulations. Still, many CUDA implementations exist for Finite Difference (FD) and Finite Element (FEM) methods for solving partial differential equations (PDEs) and some papers discuss implementations of the Navier-Stokes equation, e.g. [13], with good results. The measured speedups, comparing the GPU with a single core of a CPU, are very satisfactory and we may assume that ocean as well as atmosphere simulations can enjoy performance boosts when employing a GPU.

In this thesis we will explore the possibilities for accelerating a finite difference method for solving a model of the barotropic quasi-geostrophic vorticity circulation. The model describes the homogeneous wind-driven ocean circulation and is one of the cornerstone models in modern physical oceanography.

We will discuss, after a brief introduction to CUDA, the model and numerical methods in detail. Solving the model consists largely of two parts, computing the discretization using a finite difference scheme and solving the Poisson equation using a fast Poisson solver. Numerical algorithms in C for these methods are ported to CUDA. Implementation details and optimization steps are then discussed to show the coding style and difficulties that may arise when programming for the GPU. The last chapters will give an overview of the numerical results and the achieved performance increase.

Chapter 1

Introduction to CUDA

The CUDA programming language is tailor-made for the NVIDIA GPU and therefore has some restrictions based on the hardware layout. This leads to the best performance, but it undermines portability. As a result of this, it is important to understand the hardware layout in detail.

Since the organization of threads and the hardware layout are closely connected we will discuss both subjects at once. We refer to the CPU as the *host* and the GPU as the *device*.

1.1 Hardware layout and thread organization

A thread is an instance of software that runs on a processing unit and can access and modify memory. The CPU can handle several threads in parallel depending on the number of cores and the hardware specifications. If more threads are active, they are scheduled by the scheduler of the operating system. The scheduler activates and deactivates threads, such that they appear to run in parallel, which is not the case. In this case we say that the threads are running *concurrently*. If the scheduler is smart, idle time of a thread can be hidden by deactivating it and instead activating another thread.

The building block of the GPU is a *multiprocessor* (MP). It consists of eight cores each capable of running four threads in parallel. An MP can therefore handle 32 threads at once; this is called a *warp*. The clock frequency of a core is in the order of 1-1.5 GHz, which is low compared to a CPU. A GPU has multiple MPs and a modern CUDA enabled GPU may have more than 500 cores. Figure 1.1 shows a schematic representation of the GPU; the memory types will be discussed later.

With these large amounts of threads that can be run in parallel, it is possible to assign one thread per data element. For example with matrix addition, one thread can compute one element of the resulting matrix. By this one-to-one correspondence between a thread and a data element you soon run out of available cores to run the threads on. Therefore they have to be scheduled and wait for other threads to finish. For this reason the threads have been organized into thread blocks, which is the smallest number of threads to be scheduled on a MP.

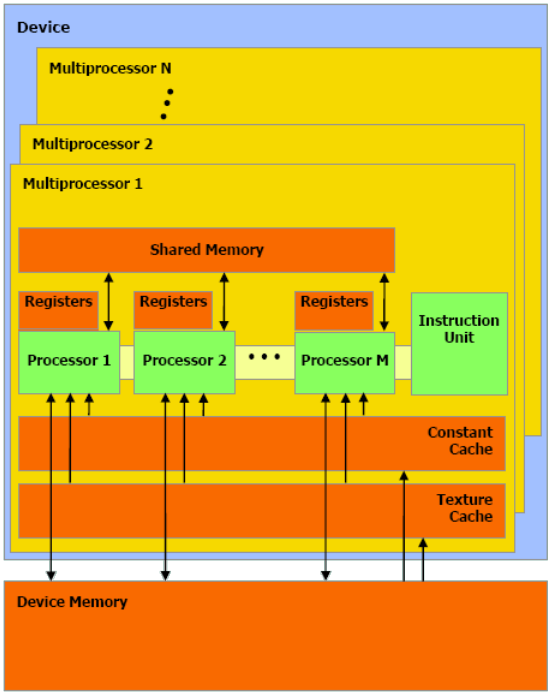


Figure 1.1: Hardware layout of a GPU [8].

1.1.1 Thread model

A *kernel* is a piece of code that is executed by each thread. Since a thread has its own threadID, we can apply the same kernel to different data. This idea is called Single Instruction Multiple Threads (SIMT). The organization of threads helps to achieve work balance and it defines the one-to-one correspondence between threads and data.

The first level of organization is the thread block. It has three dimensions and has a maximum size of $(512 \times 512 \times 64)$. Blocks are groups of threads and are the smallest

elements which can be scheduled to a multiprocessor. The blocks are divided among the multiprocessors in a cyclic fashion. This results in a near-optimal work balance since each MP gets approximately the same number of threads. Of course this does depend on the choice of the block size and the problem size.

Threads inside a block of size blockDim.y×blockDim.x get a three-dimensional local threadID,

$$\text{threadID} = \text{threadIdx.z} * \text{blockDim.y} * \text{blockDim.x} + \text{threadIdx.y} * \text{blockDim.x} + \text{threadIdx.x}.$$

The maximum number of threads in a block depends on the hardware compute capability [1]. Each block has the same number of threads and these threads have the same local numbering.

Blocks are organized into a grid which has one or two dimensions and each block has a blockID: (blockIdx.y, blockIdx.x). A grid is used to identify threads with data elements and it allows to compute the global index of a thread. For example, a matrix of size 32×32 can be divided in four blocks of size 16×16. These blocks can then be handled by a thread block of size 16×16. The threads have a local numbering from $0, \ldots, 15$, both in the x- and y-direction. Using the blockIDs we can compute the global indices, thread (threadIdx.y, threadIdx.x) will process element a_{ij}, with

$$\begin{aligned} i &= \text{blockIdx.y} * \text{blockDim.y} + \text{threadIdx.y} \\ j &= \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x}. \end{aligned}$$

Figure 1.2 illustrates the structure of a CUDA program. One host thread launches the program and executes serial code. When a kernel is launched it spawns a grid of threads on the GPU. The number of thread blocks and threads per blocks can be configured by a special syntax,

$$\text{kernel\_name} <<< \text{dimGrid}, \text{dimBlock} >>> (\text{parameters});$$

[1] The compute capability is a number of two digits, x.y, where x is the major revision number of the hardware. This is only increased if a new generation of GPUs is introduced like the NVIDIA Fermi, which has major revision number 2. The minor revision number y determines some hardware specifications. For example, devices with compute capability x.1 or higher have relaxed conditions for coalesced memory access.

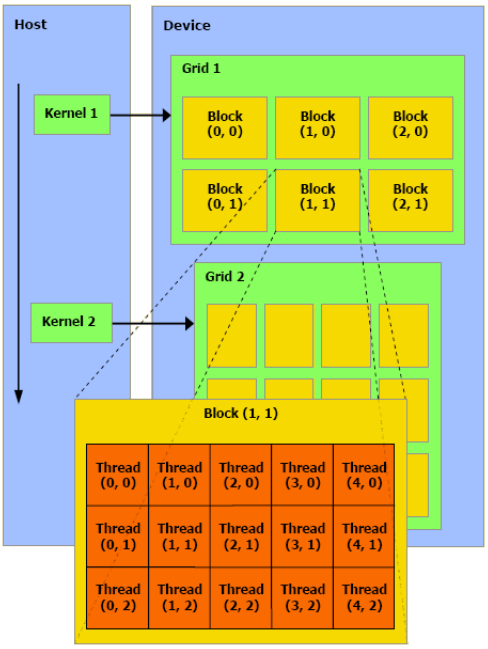


Figure 1.2: Structure of a CUDA program [8].

The structures dimGrid and dimBlock contain the dimensions of the thread grid and thread block respectively.

When the blocks have been scheduled to the MPs, the MP activates the threads of several blocks up to 1024 threads (depending on the compute capability). These blocks are divided in warps of 32 threads with consecutive threadIDs. The warps run concurrently; whenever a warp is idle another warp will run.

1.1.2 Memory model

The GPU has three basic levels of memory: the *global memory* (or device memory), *shared memory* and thread registers, see Figure 1.1. Global memory is shared among all threads, but has a much higher latency than shared memory or registers. Shared memory and registers both have a similar low latency, but are only visible to a subset of the threads.

The global memory has a typical size of several gigabytes and is used to hold the dataset of the algorithm. Its main advantage is this large size which allows to use only GPU memory during an algorithm. This makes the need to communicate data between

the CPU and the GPU minimal, which is favourable performance-wise.

A regular pattern in a CUDA program is to load or create the dataset on the host, transfer it to global memory, execute the algorithm and send the result back to host memory. This pattern is repeated if necessary.

Communication between the host and the device is very slow since it involves the PCI Express BUS, which has a low bandwidth compared to global memory. Therefore communication between the host and the device should be kept at a minimum.

Shared memory is similar to a cache, but it has to be programmed manually. Each multiprocessor has one shared memory, so it is only available to threads running on that multiprocessor. This implies that the memory is shared by threads in a block. Shared memory is almost 100x faster than global memory and provides a fast way to reuse data. The size is currently 8 kB or 16 KB for compute capability 1.x and 32 kB for compute capability 2.x.

The last type of memory are registers. These are a per-thread local memory and are used to store local variables, such as the threadID and kernel variables. Although this memory is very fast it should be avoided to have too many registers per kernel. There is a limit of registers as well as the number of threads that can be simultaneously resident on a multiprocessor and this directly impacts the number of blocks that can be scheduled. For example, if the maximum number of threads per MP is 1024 and a multiprocessor has 16384 registers, then we can have the following schemes:

1. 512 threads, 2 blocks
2. 256 threads, 4 blocks
3. 128 threads, 8 blocks
4. 32 threads, 32 blocks.

However, the maximum number of resident blocks per MP is eight, so only the first three schemes are possible. With 1024 threads active on the MP, the limit for registers is $16384/1024 = 16$ per kernel.
Table 1.1 lists the lifetimes and visibility of the different memory types.
Other read-only memory types are *texture* and *constant* memory with a size of 64 kB. These have a cache of 8 kB and can be useful for fast lookup tables.

Type	Size	Visible by	Lifetime
global	4 GB	every thread	whole program
shared	16 kB	block	kernel
register	NA	thread	warp

Table 1.1: Memory size and context for NVIDIA Tesla as used in experiments.

1.2 Coalesced memory access

With so many cores, a kernel is often memory bound, i.e., memory access time is larger than the computation time. In such cases it is important to utilize the maximum available bandwidth. This is possible when global memory access is coalesced.

When global memory is accessed by a thread there is a latency of 400 to 600 clock cycles. Memory requests are served for half-warps, but generally this does not result in 16 (for a warp size of 32) times the latency of one memory operation. This is because memory access can be *coalesced*, meaning that the memory request of one half-warp is served in one or two memory operations.

Full memory coalescing is achieved when the first thread of a half-warp accesses the first data word of an aligned memory segment and each consecutive thread accesses a consecutive data word. Global memory consists of aligned segments of 16 and 32 data words. Figure 1.3 shows two 128-byte segments of linear global memory. All memory addresses have been numbered from 0 to 63. This example consists of 64 data words of 4-byte width (type `float`). When a half-warp accesses memory only in the blue or only in the orange part, it takes at most one memory operation of 128 bytes.

To clarify this we will look at some examples of access patterns in Table 1.2. An access pattern can be described by the notation $t_k \longrightarrow d_k$, $k = 0, \ldots, 15$. This means that thread t_k with local index k accesses the data word with address d_k. From these

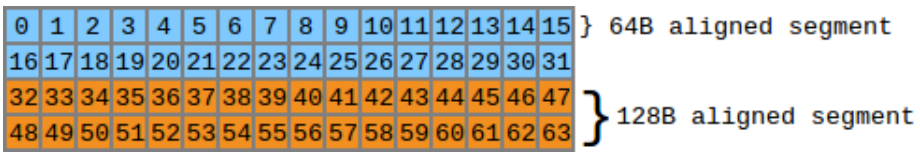


Figure 1.3: Aligned global memory segment.

examples only the first pattern is fully coalesced and all 64 bytes are transferred in a single memory operation. This results in maximum bandwidth. In the last example,

also 64 bytes are transferred in one memory operation, but 128 bytes could have been sent. This pattern uses only half of the maximum available bandwidth.

Access pattern	# memory operations	Note
$t_k \longrightarrow k$	1×64B	fully coalesced
$t_k \longrightarrow k+1$	1×128B	misaligned
$t_k \longrightarrow k+17$	1×64B and 1×32B	misaligned within two 128B segments
$t_k \longrightarrow 2k$	1×128B	access with stride of 2

Table 1.2: Access patterns for coalesced memory access, for threads of a half-warp t_k, $= 0, \ldots, 15$. E.g., $t_k \longrightarrow k$ means that thread k accesses memory address k.

1.3 Bank conflicts

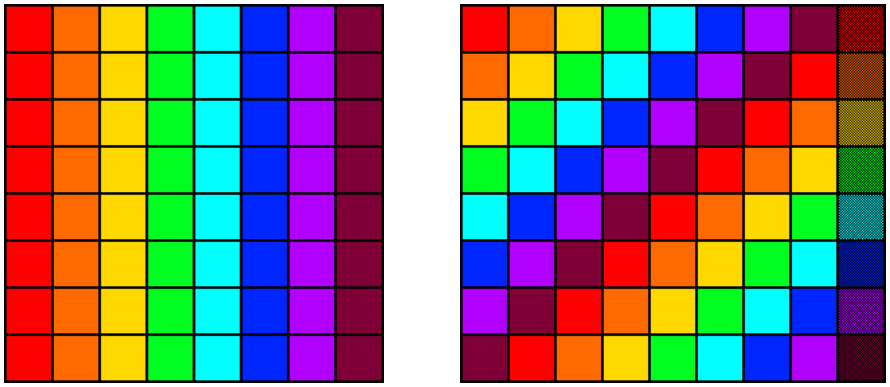
Shared memory access cannot be coalesced, instead memory can be accessed in parallel from different memory banks. Shared memory on devices with compute capability 1.x (first generation) have a size of 16 kB. The memory is divided among 16 memory banks, each consisting of 1 kB. When data is written to linear memory it is arranged such that consecutive data elements are written to separate memory banks. When data is accessed from shared memory all memory requests for separate banks are handled in parallel. If multiple threads want to access the same memory bank a bank conflict is created. The memory operations on these banks are then serialized. As for global memory access, shared memory access is handled per half-warp.

Example: matrix transposition

To illustrate this effect we discuss the example of out-of-place matrix transposition, which also has been used in our CUDA implementation of the oceanographic model. Suppose an $n \times n$ matrix A is stored in global memory as a linear array such that $A_{ij} = A[i * n + j]$. A block size of 16×16 threads is chosen such that the matrix is divided among $n/16 \times n/16$ blocks. Then row-wise memory access is coalesced.

If no shared memory is used, each thread block reads in a block of A row-wise. This row is then written column-wise to the corresponding block of the output matrix B. This results in uncoalesced writes with stride n. If the block is first transferred to shared memory, the data can be read out column-wise from shared memory and written

row-wise to B. This makes the whole memory operation coalesced, but there are still bank conflicts.



(a) Bank conflicts for column-wise access.

(b) Bank conflict free access, both row- and column-wise.

Figure 1.4: Shared memory bank conflicts can be eliminated by allocating the right amount of memory.

To see this look at Figure 1.4(a), we assume for simplicity that a block is 8×8, a half-warp has size eight and there are eight memory banks. This is a block of A written to shared memory row-wise. We have coloured each element to indicate on which memory bank it resides. It is now clear that accessing the shared memory column-wise will create an 8-way bank conflict, i.e., the whole column resides on the same memory bank and accessing this column will be serialized into eight memory operations.

The problem can be solved easily by allocating one column more in shared memory. This last column is not used for storage, but it will shift the memory banks that are accessed by the thread block. Figure 1.4(b) shows the result. Accessing a column in shared memory is now bank conflict free.

1.4 Warp divergence

As stated previously, threads are executed in groups of warps. If threads inside a warp are taking different execution paths we say the warp is *branching* or *diverging*. In this case each separate branch is computed sequentially.

Consider the following example by Mark Harris [6], which is a reduction algorithm:

```
__global__ void reduce(int *A)
{
```

```
  unsigned int tid = threadIdx.x;
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0)
      A[i] = A[i] + A[i+s];
    __syncthreads();
  }
}
```

The condition on 'tid' divides a warp in two branches, where only a small fraction of the warp does the actual work. This problem is fixed by using a strided index:

```
__global__ void reduce(int *A)
{
  unsigned int tid = threadIdx.x;
  unsigned int offset = blockDim.x + threadIdx.x;
  int i = offset + tid;

  for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2*s*tid;
    if (index < blockDim.x)
      A[offset+index] += A[offset+index + s];
    __syncthreads();
  }
}
```

Warp branching can dramatically decrease the performance of a kernel and should be avoided if possible. Branching of different warps does not involve a performance penalty, so it is possible to impose conditions on the blockID. However, the runtime of a kernel is determined by the slowest block, so all blocks should have approximately the same runtime.

Note that the for-loop ends with a synchronization, __syncthreads(). This acts as a barrier for threads in a block and has very little overhead. Global synchronization for all threads within a kernel is not possible, but a kernel launch acts as global thread barrier.

1.5 Performance benchmarking

Profiling and performance benchmarking are critical tools to optimize a program and to expose the gained performance from those optimizations. In distributed memory models the performance is split into computation time and communication time, with

communication often a bottleneck. The computation time depends on the parallelization of an algorithm and the communication time on the distribution of the data.

For GPUs there is no perfect equivalent for communication, since all data is shared between threads, if it has not been transferred to shared memory. The closest parallel is memory bandwidth, which is the rate of data that is accessed in memory per second.

The theoretical bandwidth is the peak bandwidth that can be reached and it follows from the hardware specifications. The GPU memory is double data rate (DDR) RAM and has a clock rate C and a b-bit wide memory interface. The theoretical bandwidth B^* in gigabytes per second (GB/s) is computed as

$$B^* = C \cdot 2(b/8)/10^9 \tag{1.1}$$

The device we will be using for experiments (Appendix A) has a theoretical bandwidth of

$$\begin{aligned} B^* &= (800 \cdot 10^6 \cdot (512/8) \cdot 2)/10^9 \\ &= 102.4 \text{ GB/s}. \end{aligned}$$

The factor two comes from the *double data rate* and the memory interface is divided by eight to convert to bytes. The division by 10^9 is to convert the result to GB/s.

The effective bandwidth B^e is the measured bandwidth and is simply a count of the number of memory accesses divided by the time, i.e.,

$$B^e = ((B_r + B_w)/10^9)/\text{time},$$

with B_r and B_w the number of bytes read and written per kernel respectively. The time is the wall time elapsed during kernel execution.
The ratio B^e/B^* tells us how efficient a kernel can access the memory. However, this methods has its flaws, since the kernel execution time also includes the computation time and therefore $B^e < B^*$.

Maximizing the effective bandwidth is a very important step in optimizing a kernel. In practice this comes down to designing the kernel to have fully coalesced memory access. This limits the access pattern of the threads to a simple linear one, where each consecutive thread accesses a consecutive data element. Usage of shared memory can also significantly improve the effective bandwidth.

The choice of the algorithm is often the key to achieve high performance. Sometimes a good algorithm can be found by working around the limitations of coalesced memory access and warp divergence, but often it pays to find an alternative, more suitable algorithm.

The computational performance largely depends on the work balance, which is attained by the cyclic distribution of blocks to multiprocessors. It is possible that the problem size is small and that there are less blocks than MPs. Decreasing the block size will then improve the performance.

The occupancy of a multiprocessor should also be as large as possible. The *occupancy* is defined as the number of active threads on an MP. As we have discussed earlier, this depends on the register pressure of a kernel. Therefore it pays to have small kernels with few registers and to find the optimal block size.

For benchmarking a CUDA program several options are available. NVIDIA has provided a profiler, which shows the occupancy and run times of individual kernels. The program is not very mature and the command line interface is not very usable. The graphical user interface can be helpful, but only if one has direct access to the machine containing the GPU. Other good profilers exist, but they are proprietary and expensive.

To measure run times one can use the standard timers provided by the C programming language. Kernel calls are asynchronous, so after a kernel launch the control immediately returns to the CPU. When benchmarking a kernel one should use the cudaThreadSynchronize() function which acts as a barrier for the host thread and waits until the device has completed all preceding requested tasks.

Chapter 2

The model

2.1 Barotropic vorticity equation

The theory of the homogeneous wind-driven ocean circulation is one of the cornerstones in modern physical oceanography. Consider a rectangular ocean basin of size $\pi L \times \pi L$ having a constant depth D. The basin is situated on a mid-latitude β-plane with a central latitude $\theta_0 = 45°\text{N}$ and Coriolis parameter $f_0 = 2\Omega \sin\theta_0$, where Ω is the rotation of the Earth. The meridional variation of the Coriolis parameter at the latitude θ_0 is indicated by β_0. The density ρ of the water is constant and the flow is forced at the surface through a wind-stress vector $\mathbf{T}_* = (\tau_*^x(x,y), \tau_*^y(x,y))$ where the *-subscript indicates dimensional quantities. Flows in the basin are governed by the barotropic vorticity equation and its dimensional form (with $\psi_* = p_*/(\rho_0 f_0)$ in m^2s^{-1}) is

$$\left(\frac{\partial \psi_*}{\partial t_*} + \frac{\partial \psi_*}{\partial x_*}\frac{\partial}{\partial y*} - \frac{\partial \psi_*}{\partial y_*}\frac{\partial}{\partial x_*}\right)\left(\nabla_*^2\psi_* - \lambda_0\psi_*\right) + \beta_0\frac{\partial \psi_*}{\partial x_*} = \frac{f_0}{D}w_{E*} - \epsilon_0\nabla_*^2\psi_* + A_H\nabla_*^4\psi_*, \quad (2.1)$$

where $\epsilon_0 = f_0\delta_E/D(s^{-1})$ is a dimensional damping coefficient and $\lambda_0 = f_0^2/(gD) = 1/R_D^2$, where R_D is the external Rossby radius of deformation and A_H is the lateral friction coefficient. The Ekman upwelling velocity w_{E*} is given by

$$w_{E*} = \frac{1}{\rho f_0}\left(\frac{\partial \tau_*^y}{\partial x_*} - \frac{\partial \tau_*^x}{\partial y_*}\right). \quad (2.2)$$

The governing equations are non-dimensionalized using a horizontal length scale L, a vertical length scale D, a horizontal velocity scale U, a streamfunction scale UL, the advective time scale L/U and a characteristic amplitude of the wind stress, τ_0. The

effect of deformations of the ocean-atmosphere interface on the flow is neglected ($\lambda_0 \approx 0$). The dimensionless barotropic quasi-geostrophic model of the flow for the dimensionless barotropic streamfunction ψ can be written as

$$\left[\frac{\partial}{\partial t} + u\frac{\partial}{\partial x} + v\frac{\partial}{\partial y}\right]\left[\nabla^2\psi + \beta y\right] = -\mu\nabla^2\psi + \alpha\left(\frac{\partial \tau^y}{\partial x} - \frac{\partial \tau^x}{\partial y}\right), \tag{2.3}$$

where the horizontal velocities are given by $u = -\partial\psi/\partial y$ and $v = \partial\psi/\partial x$. The parameters in (2.3) are the planetary vorticity gradient β, the wind-stress strength α and the bottom friction coefficient μ, given by

$$\alpha = \frac{\tau_0 L}{\rho D U^2}; \quad \beta = \frac{\beta_0 L^2}{U}; \quad \mu = \frac{\epsilon_0 L}{U}. \tag{2.4}$$

Kinematic conditions on all lateral boundaries are $\psi = 0$. The wind-stress forcing pattern is prescribed as

$$\tau^x(x, y) = -\frac{1}{2}\cos 2y, \tag{2.5}$$

$$\tau^y(x, y) = 0, \tag{2.6}$$

which is symmetric with respect to the mid-axis of the basin (the standard double-gyre case).

By introducing the vorticity $\zeta = \nabla\psi$ we can write (2.3) as a system of equations,

$$\begin{cases} \dfrac{\partial\zeta}{\partial t} = -\mu\zeta + \mathrm{Re}^{-1}\Delta\zeta + \dfrac{\partial\psi}{\partial y}\dfrac{\partial\zeta}{\partial x} - \dfrac{\partial\psi}{\partial x}\left(\dfrac{\partial\zeta}{\partial y} + \beta\right) - \alpha\sin(2y) \\ \quad \zeta = \Delta\psi, \end{cases} \tag{2.7}$$

with Re the Reynolds number, which is inversely proportional to the lateral friction coefficient A_H,

$$\mathrm{Re} = \frac{UL}{A_H}.$$

The boundary conditions and initial condition are

$$\left.\begin{aligned} \psi(t, x, y) &= 0, \\ \zeta(t, x, y) &= 0, \end{aligned}\right\} \text{for } (x, y) \in \partial\Omega \tag{2.8}$$

$$\psi(0, x, y) = 0, \qquad \text{for } (x, y) \in \Omega. \tag{2.9}$$

2.2 Discretization

To the dimensionless model we apply a finite difference method. The choice for the discretization scheme has an important impact on the performance of the GPU program. There are several aspects we must consider. First of all memory access needs to be coalesced to achieve maximal bandwidth. Basically this means that the stencil should have adjacent points in the spatial dimension.

For the reason of coalesced memory access we choose the central finite difference scheme in the space direction, but most other schemes are suitable too. For time integration we choose the Adams-Bashforth linear multistep method [3], since this is a regular choice for this model.

We consider a square domain $\Omega = [0, \pi] \times [0, \pi]$ which is divided in a grid with $N_x \times N_y$ grid points:

$$(x_i, y_j), \qquad i = 0, 1, \ldots, N_x - 1, \qquad j = 0, 1, \ldots, N_y - 1. \tag{2.10}$$

In the x-direction the step size is $\Delta x = \frac{L}{N_x - 1}$ and in the y-direction it is $\Delta y = \frac{L}{N_y - 1}$. We choose an equal step size in both directions $h = \Delta x = \Delta y$ and $N = N_x - 2 = N_y - 2$ which simplifies the notation, but it will also allow us to employ a fast Poisson solver for the second equation in (2.7).

We use the following notation for the unknowns evaluated on the grid:

$$\psi_{ij}^n = \psi(t_n, x_i, y_j) = \psi(n\Delta t, i\Delta x, j\Delta y).$$

With these coordinates the spatial discretization of the first equation of (2.7) is written in matrix notation as:

$$F = (\Psi B)(BZ) - B\Psi\,(ZB + \beta I_N) + \frac{1}{\mathrm{Re}}\,(CZ + ZC) + \mu I_N Z - T, \tag{2.11}$$

with boundary and initial conditions

$$\psi_{0j} = \psi_{i,N+1} = 0, \tag{2.12}$$

$$\zeta_{0i} = \zeta_{i,N+1} = 0, \tag{2.13}$$

$$\psi_{i,j}^0 = 0, \qquad \text{for} \quad i, j = 0, \ldots, N + 1 \tag{2.14}$$

where the unknowns of the interior of the grid are contained in

$$\Psi_{ij} = \psi_{i+1,j+1}, \quad Z_{ij} = \zeta_{i+1,j+1}, \qquad i, j = 0, \ldots, N - 1,$$

with I_N the $N \times N$ identity matrix.

The matrices B and C follow from the standard central finite difference approximation with local truncation error $\mathcal{O}\left(h^2\right)$,

$$B = \frac{1}{2h}\begin{bmatrix} 0 & -1 & & & & \\ 1 & 0 & -1 & & & \\ & 1 & 0 & -1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 0 & -1 \\ & & & & 1 & 0 \end{bmatrix}, \quad C = \frac{1}{h^2}\begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & 1 & -2 & 1 & \\ & & & 1 & -2 & \end{bmatrix}, \tag{2.15}$$

$$T = \alpha \begin{bmatrix} \sin 2y_1 & \dots & \sin 2y_N \\ \vdots & & \vdots \\ \sin 2y_1 & \dots & \sin 2y_N \end{bmatrix} \tag{2.16}$$

and T is the matrix containing the discrete wind-stress forcing patterns.

For time integration, the Adams-Bashforth method is used. Given is the ordinary differential equation (ODE) that arises from the discretization in space of (2.7):

$$\dot{\zeta} = f(t, \zeta), \quad \zeta(t_0) = \zeta^0,$$

here $f_{ij} = F_{ij}$. We use the two-step method:

$$\zeta^{n+2} = \zeta^{n+1} + \Delta t \left(\frac{3}{2} f(t_{n+1}, \zeta^{n+1}) - \frac{1}{2} f(t_n, \zeta^n) \right). \tag{2.17}$$

Note that this method uses two time steps from the past, while the ODE has only one initial condition y_0. To solve this problem we can use Euler Forward to compute y_1 first and then continue with Adams-Bashforth. For the model, we assume that $\psi^0 = 0$, i.e., we assume that the wind starts blowing at $t = 0$. Therefore we simply define $\psi^{-1} \equiv 0$.

In matrix notation the full discretization of (2.7) is given by

$$Z^{n+2} = Z^{n+1} + \frac{\Delta t}{2} \left(3F^{n+1} - F^n \right). \tag{2.18}$$

These schemes are used only for the first equation of (2.7), for the second equation we use the standard five-point stencil using central finite differences. This leads to a system of equations that can be solved using a fast Poisson solver.

2.3 Poisson solver

Since we have split the PDE model in a system of two equations, only the first involves time integration. To extract the solution ψ^{n+1} from ζ^{n+1} we have to solve the Poisson equation:

$$\Delta\psi = \zeta, \tag{2.19}$$

if $\zeta = 0$ we call (2.19) the Laplace equation.

The discretization of (2.19) is given by

$$A\psi = \zeta, \tag{2.20}$$

where $-A$ is a square, positive definite, symmetric matrix resulting from the discretization of the Laplace operator with the usual 5-point approximation (in 2D, or 3-point stencil in 1D):

$$\zeta_{ij} \approx \frac{\psi_{i+1,j} - 2\psi_{ij} + \psi_{i-1,j}}{(\Delta x^2)} + \frac{\psi_{i,j+1} - 2\psi_{ij} + \psi_{i,j-1}}{(\Delta y^2)}. \tag{2.21}$$

Matrix A has size $J \times J$ with $J = N^2$, so the grid is stored in a vector of length J. This represents the data structure that is used in the implementation. The unknowns are stored row-wise in a vector, such that grid points are consecutively in the y-direction. For example ψ is stored as:

$$\psi = [\psi_{11}, \zeta_{12}, \ldots, \psi_{1N}, \psi_{21}, \ldots, \psi_{2N}, \ldots, \psi_{N1}, \ldots, \psi_{NN}].$$

The discretization of (2.19) is now written in block-matrix notation as

$$\frac{1}{h^2}\begin{bmatrix} K-2I & I & & & & & \\ I & K-2I & I & & & & \\ & I & K-2I & I & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & I & K-2I & I & \\ & & & & I & K-2I \end{bmatrix} \psi = \zeta, \tag{2.22}$$

with

$$K = \begin{bmatrix} -2 & 1 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & -2 & 1 \\ & & & & 1 & -2 \end{bmatrix}$$

an $N \times N$ matrix.

The eigenvectors and eigenvalues of A are known and we can use them to solve the linear system. The eigenvectors are given by

$$y_{kl}(i,j) = \sin\frac{ik\pi}{N+1}\sin\frac{jl\pi}{N+1}, \tag{2.23}$$

with corresponding eigenvalues [12],

$$\lambda_{kl} = -4 + 2\cos\frac{k\pi}{N+1} + 2\cos\frac{l\pi}{N+1}, \tag{2.24}$$

for $k, l, i, j = 1, \ldots, N$.

It can be proven that the eigenvectors form an orthogonal basis of $\mathbb{R}^N$, so we can write ζ in the basis of eigenvectors:

$$\zeta = \sum_{k=1}^{N}\sum_{l=1}^{N} c_{kl} y_{kl}, \tag{2.25}$$

with c_{kl} the coefficients.

Next we scale the coefficients c_{kl} by the eigenvalues λ_{kl}. We can then reconstruct the solution ψ by

$$\psi = \sum_{k=1}^{N}\sum_{l=1}^{N} \frac{c_{kl}}{\lambda_{kl}} y_{kl}. \tag{2.26}$$

To see that this is really the solution we multiply (2.26) by A from the left

$$\begin{aligned} A\psi &= \sum_{k=1}^{N}\sum_{l=1}^{N} \frac{c_{kl}}{\lambda_{kl}} A y_{kl} \\ &= \sum_{k=1}^{N}\sum_{l=1}^{N} \frac{c_{kl}}{\lambda_{kl}} \lambda_{kl} y_{kl} \\ &= \sum_{k=1}^{N}\sum_{l=1}^{N} c_{kl} y_{kl} \\ &= \zeta. \end{aligned}$$

In the second step we used that y_{kl} is an eigenvector of A.

A closer look at the eigenvectors tells us that (2.25) and (2.26) are in fact two-dimensional discrete sine transforms (DSTs). Note that the one dimensional DST-I sine transform $\{X_k\}$ of a sequence of numbers $\{x_k\}$ is given by

$$X_k = \sum_{j=1}^{N} x_j \sin\left(\frac{\pi}{N+1} jk\right), \quad k = 1, \ldots, N. \tag{2.27}$$

Instead of actually computing the matrix products of $\mathcal{O}\left(N^2\right)$ complexity we can use a fast algorithm for computing the sine transform. Ideally this would be an algorithm of $\mathcal{O}\left(N \log_2 N\right)$ complexity as proposed by Püschel and Moura [11], but implementing these methods requires a substantial amount of work and FFT-based (Fast Fourier Transform) algorithms for the sine transform with the same asymptotic complexity, but with a higher constant (2x) exist too. These algorithms can benefit from optimized implementations of the FFT.

Solving (2.20) consists of three steps:

1. Compute the coefficients c_{kl} using the inverse 2D DST-I applied to ζ,
2. Divide the coefficients by the eigenvalues: $c_{kl} \mapsto \frac{c_{kl}}{\lambda_{kl}}$,
3. Construct ψ by performing again a 2D DST-I on the coefficients c_{kl}.

Note that (2.27) is unscaled, which means that its inverse is scaled by a factor $2/(N+1)$. Instead of applying the inverse DST, we apply (2.27) both in step 1 and 3. The scaling is then applied afterwards.

It is clear that most of the work of the Poisson solver consists of computing the DST-I. Therefore we will focus on finding an efficient algorithm for the DST-I.

2.3.1 FFT-based sine transform

NVIDIA has a library that implements the Fast Fourier Transform in CUDA. It is based on the FFTW library, *The Fastest Fourier Transform in the West*. Unfortunately it only implements the Discrete Fourier Transform (DFT) and no real transforms like the sine or cosine transforms. However, we can still use it after a pre-process, since a sine transform contains the imaginary parts of a similar DFT.

The (non-normalized) DST-I of a sequence of numbers $x_0, x_1, \ldots, x_{N-1}$ as computed by CUFFT is defined as follows:

$$X_k = 2 \sum_{j=0}^{N-1} x_j \sin\left(\frac{\pi(j+1)(k+1)}{N+1}\right), \qquad k = 0, 1, \ldots, N-1. \tag{2.28}$$

By using the complex identity of the sine,

$$\sin x = \frac{e^{ix} - e^{-ix}}{2i}$$

we can write (2.28) as:

$$\begin{aligned} X_k &= \sum_{j=0}^{N-1} x_j \frac{e^{\frac{i\pi(j+1)(k+1)}{N+1}} - e^{-\frac{i\pi(j+1)(k+1)}{N+1}}}{i} \\ &= i\sum_{j=0}^{N-1} x_j e^{-\frac{i\pi(j+1)(k+1)}{N+1}} - i\sum_{j=0}^{N-1} x_j e^{\frac{i\pi(j+1)(k+1)}{N+1}} \\ &= i\sum_{\tilde{j}=1}^{N} x_{\tilde{j}-1} e^{-\frac{i\pi\tilde{j}\tilde{k}}{N+1}} - i\sum_{\tilde{j}=1}^{N} x_{\tilde{j}-1} e^{\frac{i\pi\tilde{j}\tilde{k}}{N+1}}, \quad \tilde{k} = 1, \ldots, N \end{aligned} \tag{2.29}$$

using $\tilde{k} = k + 1$. Now we add a zero in front of x, i.e., $\tilde{x} = [0\ x]$; and we drop the ˜'s:

$$\begin{aligned} X_k &= i\sum_{j=0}^{N} x_j e^{-\frac{i\pi jk}{N+1}} - i\sum_{j=0}^{N} x_j e^{\frac{i\pi jk}{N+1}} \\ &= i\sum_{j=0}^{N} x_j e^{-\frac{2\pi ijk}{2(N+1)}} - i\sum_{j=0}^{N} x_j e^{\frac{2\pi ijk}{2(N+1)}}. \end{aligned} \tag{2.30}$$

Another $N + 1$ zeros are appended to x, $x = [x\ 0\ \ldots\ 0]$ and (2.30) becomes

$$X_k = i\sum_{j=0}^{M-1} x_j e^{-\frac{2\pi ijk}{M}} - i\sum_{j=0}^{M-1} x_j e^{\frac{2\pi ijk}{M}}, \tag{2.31}$$

where $M = 2(N + 1)$.

The Discrete Fourier Transform is given by:

$$X_k = \sum_{j=0}^{N-1} x_j e^{-\frac{2\pi jki}{N}},$$

and its non-normalized inverse:

$$X_k = \sum_{j=0}^{N-1} x_j e^{\frac{2\pi jki}{N}}.$$

With this information (2.31) reduces to

$$X = i \cdot \mathrm{DFT}(x) - i \cdot \mathrm{inverseDFT}(x), \tag{2.32}$$

(here the inverse DFT is non-normalized) this should be a real vector, since the sine transform produces a real vector. Therefore, the imaginary terms should cancel out.

$$X = \mathcal{I}\left[\mathrm{inverseDFT}(x) - \mathrm{DFT}(x)\right].$$

We can use the following identity of the inverse DFT:

$$\mathcal{F}^{-1}\left(\{x_k\}\right) = \mathcal{F}\left(\{x_{N-k}\}\right)/N,$$

so we can compute the DST-I as one DFT of length M,

$$X = \mathrm{DFT}\left([0\ x_0\ x_1\ \ldots\ x_{N-1}\ 0\ -x_{N-1}\ -x_{N-2}\ \ldots\ -x_1\ -x_0]\right).$$

Note that we still need to scale the output by $1/2$, for the inverse DST-I we need to scale by $1/(N+1)$.

2.4 Implementation details of the Poisson solver

The most computationally intensive part of the Poisson solver is the two dimensional discrete sine transform. As we have seen, it can be computed using the Fast Fourier Transform. The advantage of this is the fact that a lot of work has been put in creating a parallel FFT for the GPU (CUFFT). In general, an implementation of the FFT is one of the first things to be ported to new parallel platforms. This adds a certain degree of portability to the program.

By using an FFT-based sine transform we do lose some performance. First of all, the FFT needs an input signal of length $M = 2(N+1)$, which approximately doubles the complexity of the algorithm compared to a DST algorithm of complexity $\mathcal{O}\left(N \log_2 N\right)$. Since the grid is stored in linear memory the grid needs to be copied out-of-place and should be extended as described previously. Below is an example for a 5×5 grid:

$$\begin{pmatrix} \zeta_{11} & \zeta_{12} & \zeta_{13} \\ \zeta_{21} & \zeta_{22} & \zeta_{23} \\ \zeta_{31} & \zeta_{32} & \zeta_{33} \end{pmatrix} \mapsto \begin{pmatrix} 0 & \zeta_{11} & \zeta_{12} & \zeta_{13} & 0 & -\zeta_{13} & -\zeta_{12} & -\zeta_{11} \\ 0 & \zeta_{21} & \zeta_{22} & \zeta_{23} & 0 & -\zeta_{23} & -\zeta_{22} & -\zeta_{21} \\ 0 & \zeta_{31} & \zeta_{32} & \zeta_{33} & 0 & -\zeta_{33} & -\zeta_{32} & -\zeta_{31} \end{pmatrix}.$$

Moreover, each row should be converted to a complex vector. The CUFFT library provides a cufftComplex type, which is simply a structure with real and complex parts interleaved. So a real vector in complex form is the real vector interleaved with zeros. A copy operation therefore writes the matrix row-wise to a matrix of type cufftComplex with a stride of two. This introduces a performance penalty of approximately 50% of the theoretical bandwidth for devices with compute capability 1.2 or higher, as shown in Figure 3.9 of [7]. Devices with lower compute capability have an even lower performance. This is a disadvantage when working with complex types. Fortunately, there is a way to

avoid copying complex data types altogether by using a real FFT algorithm. This real FFT takes as input a complex vector of half the length that is built up as

$$y_k = x_{2k} + x_{2k+1}i, \quad k = 0, 1, \ldots, \frac{N}{2} - 1, \tag{2.33}$$

where $y \in \mathbb{C}$ is the complex input vector derived from the real input vector $x \in \mathbb{R}$. Note that this allows us to copy the real matrix to a complex matrix of half the row-size without strided memory writes.

2.4.1 Real FFT

From now on we will use the convention to address the DFT of a signal by a capital letter, e.g., $\mathrm{DFT}(f) = F$.
For a purely real signal f and a purely imaginary signal g the following identities hold:

$$\left.\begin{array}{ll} F_{N-k} & = (F_k)^* \\ G_{N-k} & = -(G_k)^* \end{array}\right\} \quad k = 0, 1, \ldots, \frac{N}{2} - 1. \tag{2.34}$$

By computing a standard complex DFT the last $\frac{N}{2} - 1$ coefficients are redundant. This not only wastes computation, but it is also inefficient storage-wise. Fortunately there exists a method to compute a DFT of a real sequence without this redundancy.

Numerical recipes in C [10] presents an implementation for combining two DFTs in one. We will derive this algorithm theoretically and show how it can be used to speed up the Poisson solver.

Two sequences of n purely real numbers f and g are given and we want to compute their DFT. First we combine these signals in one complex signal, also of length n, where f is stored as the real parts and g as the imaginary parts, i.e., the new signal h is build up as:

$$h_j = f_j + g_j i, \quad j = 0, 1, \ldots, n-1.$$

This is the packing phase. After packing the data the FFT is applied to h. The result is

$$\begin{aligned} H_k &= \sum_{j=0}^{n-1} h_j e^{\frac{-2\pi i j k}{n}}, \\ &= \sum_{j=0}^{n-1} (f_j + g_j i)\, e^{\frac{-2\pi i j k}{n}}, \\ &= \sum_{j=0}^{n-1} f_j e^{\frac{-2\pi i j k}{n}} + i \sum_{j=0}^{n-1} g_j e^{\frac{-2\pi i j k}{n}}. \end{aligned} \tag{2.35}$$

From (2.34) and (2.35) we get a hint on how to proceed.
Note that

$$\mathcal{R}[H_k] = \sum_{j=0}^{n-1} \left[f_j \cos \frac{-2\pi ijk}{n} - g_j \sin \frac{-2\pi jk}{n} \right],$$
$$\mathcal{I}[H_{n-k}] = \sum_{j=0}^{n-1} \left[f_j \sin \frac{-2\pi ijk}{n} + g_j \cos \frac{-2\pi ijk}{n} \right].$$

Now we compute

$$\begin{aligned}
\mathcal{R}[H_k] + \mathcal{R}[H_{n-k}] = & \\
&= \sum_{j=0}^{n-1} \left[f_j \cos \frac{-2\pi jk}{n} - g_j \sin \frac{-2\pi jk}{n} \right] \\
&\quad + \sum_{j=0}^{n-1} \left[f_j \cos \frac{-2\pi j(n-k)}{n} - g_j \sin \frac{-2\pi j(n-k)}{n} \right], \\
&= 2 \sum_{j=0}^{n-1} f_j \cos \frac{-2\pi jk}{n}.
\end{aligned}$$

This gives us the real part of F_k. If we repeat the above computations for other combinations of the real and imaginary parts of H_k and H_{n-k} we find the following identities:

$$\begin{aligned}
\mathcal{R}[F_k] &= \frac{1}{2} \left(\mathcal{R}[H_k] + \mathcal{R}[H_{n-k}] \right), \\
\mathcal{I}[F_k] &= \frac{1}{2} \left(\mathcal{I}[H_k] - \mathcal{I}[H_{n-k}] \right), \\
\mathcal{R}[G_k] &= -\frac{1}{2} \left(\mathcal{I}[H_k] + \mathcal{I}[H_{n-k}] \right), \\
\mathcal{I}[G_k] &= -\frac{1}{2} \left(\mathcal{R}[H_k] - \mathcal{R}[H_{n-k}] \right).
\end{aligned}$$

Algorithm 1 lists the final algorithm.

This algorithm can also be used to compute one real DFT of length n as a DFT of length $n/2$. We set the odd coefficients as the real part of the input signal and the even coefficients as the imaginary parts

$$h_j = f_{2j} + i f_{2j+1}. \tag{2.36}$$

In the implementation this is simply a cast from the float type to the cufftComplex type, so no real packing has to be done. If this signal is transformed and unpacked as in

Algorithm 1 Combining two DFTs of purely real signals.

Input: Real signals f and g
Output: Fourier transforms F and G

Packing phase
for $k = 0$ **to** $n - 1$ **do**
  $h_k = f_k + g_k i$
end for

Compute Fourier transform H of h

Extract phase
for $k = 0$ **to** $n - 1$ **do**
  $F_k = \frac{1}{2}\left(\mathcal{R}\left[H_k\right] + \mathcal{R}\left[H_{n-k}\right]\right) + \frac{1}{2}\left(\mathcal{I}\left[H_k\right] - \mathcal{I}\left[H_{n-k}\right]\right) i$
  $G_k = -\frac{1}{2}\left(\mathcal{I}\left[H_k\right] + \mathcal{I}\left[H_{n-k}\right]\right) - \frac{1}{2}\left(\mathcal{R}\left[H_k\right] - \mathcal{R}\left[H_{n-k}\right]\right) i$
end for

Algorithm 2 Real DFT

Input: Real signal f
Output: Fourier transform F

Packing phase
for $k = 0$ **to** $n/2 - 1$ **do**
  $h_k = f_{2k} + i f_{2k+1}$
end for

Compute Fourier transform H of h

Extract phase
for $k = 0$ **to** $n/2 - 1$ **do**
  $F_k = H_k - \frac{1}{2}\left(1 + ie^{\frac{2\pi i j k}{N/2}}\right)\left(H_k - H^*_{N/2-k}\right), \quad k = 0, \ldots, N/2$
end for
with $H_{\frac{N}{2}} = H_0$.

Algorithm 1 we do not get the DFT of the original signal, since we have split the DFT in two. Therefore we need to adjust the extract phase.

The transformed signals contain more or less half of the DFT of the original signal f,

$$F_k^e = \sum_{j=0}^{n/2-1} f_{2j} e^{\frac{-2\pi ijk}{n/2}},$$
$$F_k^o = \sum_{j=0}^{n/2-1} f_{2j+1} e^{\frac{-2\pi ijk}{n/2}}.$$

We call F^e the even signal and F^o the odd signal. The *even* signal $F_k^e = \sum_{j=0}^{n/2-1} f_{2j} e^{\frac{-2\pi i(2j)k}{n}}$ really contains the even part of the transform of the original signal. The F^o signal does not contain the odd parts, but it does almost. First we need to multiply it by a factor $e^{i\theta}$, for some θ,

$$e^{i\theta} F_k^o = \sum_{j=0}^{n/2-1} f_{2j+1} e^{i\theta - \frac{2\pi ijk}{n/2}}. \tag{2.37}$$

Then we want

$$i\theta - \frac{2\pi ijk}{n/2} = -\frac{2\pi ijk}{n}$$
$$\Rightarrow$$
$$\theta = \frac{2\pi ijk}{n}.$$

This leads to

$$F_k = F_k^e + e^{2\pi ijk/n} F_k^o. \tag{2.38}$$

Note that $H_k = F_k^e + iF_k^o$ is the Fourier transform of the packed signal. In terms of this transform, the original Fourier transform can be written as

$$F_k = \frac{1}{2}\left(H_k + H_{N/2-k}^*\right) - \frac{i}{2}\left(H_k - H_{N/2-k}^*\right) e^{2\pi ik/N}, \quad k = 0, \ldots, N-1. \tag{2.39}$$

In the implementation we use a slightly different extract phase from [4] as suggested by Ooura. The final algorithm is listed in Algorithm 2.

This algorithm has several advantages. First of all the packing phase is trivial since it is a cast from one data type to another. Secondly, the algorithm only provides the first $N/2+1$ coefficients of the Fourier transform. Remember that computing the DST

using the FFT the input signal needs to be extended to approximately twice the length. The output holds the DST in coefficients with index 1 to N, therefore the first $N/2+1$ coefficients are all we need for the DST. This halves the amount of work.

2.5 Implementation details of the spatial finite difference approximation

We will shortly discuss the implementation of (2.11). Again the grid is divided over thread blocks. To improve memory bandwidth each block loads part of the grid to shared memory, but threads on the boundary need grid points from an adjacent thread block. Therefore each thread block should also load the halo points, see Figure 2.1. This is done by threads on the boundary of the block.

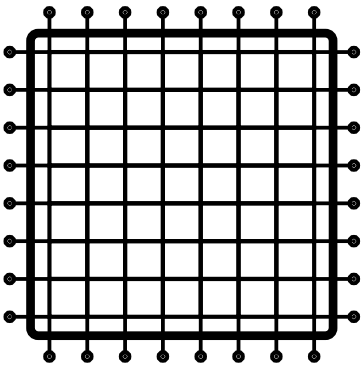


Figure 2.1: One block writes one grid block to shared memory together with its halo.

This is similar to how distributed-memory systems handle finite differences. The updated grid points are directly written to global memory, row-wise and fully coalesced. The whole procedure is in-place.

Chapter 3

Numerical results

The numerical simulations were conducted on an NVIDIA Tesla GPU with specifications as listed in Appendix A. Initially this device did not support double precision, therefore simulations on the GPU use single precision. Moreover, the results for double precision are up to eight times slower. This is because GPUs are more optimized for single precision floating point arithmetic, which is sufficiently accurate for computer graphics.

Whenever we compare the speed of the GPU with a CPU[1], we implicitly mean a single core of the Intel Xeon also listed in Appendix A.

For the experiments, parameter values are the same as used by Berloff and McWilliams [2]. Although we can not accurately compare our results, since they use 1.5- and 2-layer approximations, these parameter values will yield a stable solution,

$$L = 3840 \text{ km} \qquad \epsilon_0 = 0$$
$$f_0 = 8.3 \cdot 10^{-5} \text{ s}^{-1} \qquad \rho = 10^3 \text{ kg/m}^3$$
$$D = 300 \text{ m} \qquad \tau_0 = 5 \cdot 10^{-2} \text{ N/m}^2$$
$$\beta_0 = 2 \cdot 10^{-11} \text{ s}^{-1}\text{m}^{-1} \qquad A_H = 400 \text{ m}^2/\text{s}$$

$$U = 2.17 \cdot 10^{-3} \text{ m/s} \qquad \text{Re} = 20.83$$
$$\mu = 0, \qquad \alpha = 1.36 \cdot 10^5$$
$$\beta = 1.36 \cdot 10^5$$

[1]The sequential program is written in C and performs exactly the same tasks as the CUDA implementation. The difference is that the sequential program uses lookup tables for the Poisson solver. Also, it uses double precision, since the FFT library (FFTW) was compiled with double precision on the test system.

The dimensionless time is

$$t = \frac{U}{L} t^*.$$

A simulation of 10 years is enough to let the solution converge to a steady state. This corresponds to a dimensionless time of

$$t = 0.1783$$

This leads to very small time steps for large sized grids, but it will stay well within the range of the single precision floating-point format.

As the block size we have chosen 16×16, such that half-warps access separate rows of the grid. This is the minimal block size to enable fully coalesced memory access. Moreover, shared memory has 16 memory banks, so avoiding bank conflicts will be easier. First of all we have compared the time per iteration of the sequential program

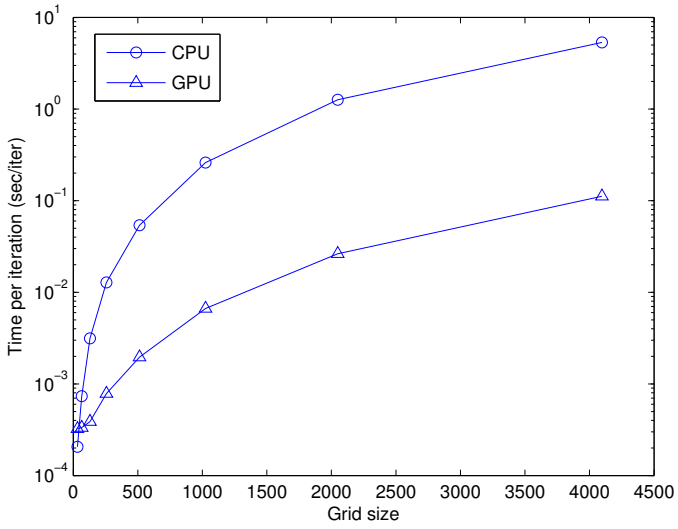


Figure 3.1: Time per iteration of CPU and GPU algorithms.

and the CUDA implementation. This gives an overview of the performance and it enables us to estimate the time needed to perform a full simulation of 10 years.

Since the time per iteration does not depend on the time step we have chosen $\Delta t = 10^{-7}$ and a final time of 0.001 for each grid size, which is a total of 10,000 iterations. The time per iteration is taken as the average over these 10,000 iterations. The results are shown in Figure 3.1. For small grids the solution diverges, since the spatial step size

has to satisfy

$$5h < \sqrt[3]{\frac{A_H}{\beta_0}},$$

though the run times are still accurate.

For a grid size of 33 the sequential program is faster, but already for a grid size of 65 the GPU wins. From this figure we see that the GPU performs very well.

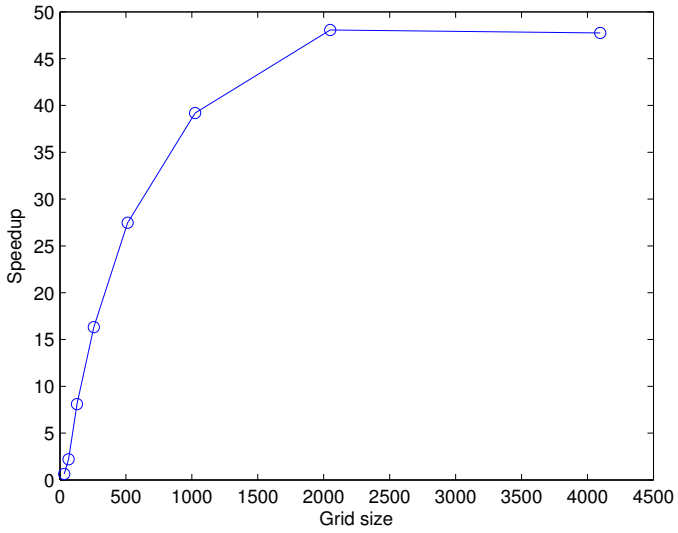


Figure 3.2: Speedup of GPU compared to CPU.

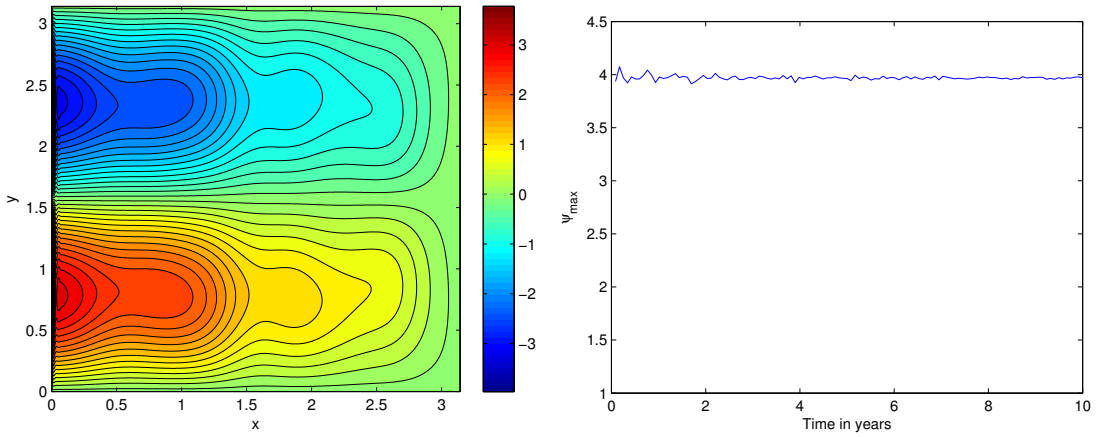
In Figure 3.2 we have plotted the speedup comparing the GPU with one core of the CPU. The speedup S is defined as

$$S := \frac{T_s}{T_p}, \tag{3.1}$$

with T_s and T_p the run times of the sequential and parallel program respectively. A speedup of 48 is reached for a grid size of 2049. For small grid sizes the speedup is less impressive. If we look closely, the speedup decreases slightly if the grid size is increased to 4097. This is probably due to the overhead caused by CUDA. At some point the speedup should stagnate, when the total number of active threads is reached. At that point, further increasing the number of threads (larger grid sizes) does not result in a larger speedup, but it will only add more overhead.

We should note that all grid sizes are such that the lengths of the Fourier transforms are a power of two. The performance of radix-2 FFTs is generally much better than

mixed radix transforms. In Appendix B more detailed tables are listed of the performance. Note the large performance drop when the grid size is only one off the ideal grid size[2].



(a) Contour plot of the flow at $t = 0.1783$

(b) Convergence history

Figure 3.3: Numerical simulation on a 1025×1025 grid for a Reynolds number of 10.

Figure 3.3 shows the results of a full simulation over 10 years. With a grid of 1025×1025 and a time step of $5 \cdot 10^{-7}$ this takes about 40 minutes. The sequential program would take over a day for such a simulation. Full simulations on grids of size 2049 and 4097 are also feasible with run times of 13 hours to several days respectively. This is the great strength of GPUs. While the speedup for small grids is moderate, simulations on larger resolutions are possible.

We have measured the run times of the Poisson solver and the grid update. The Poisson solver makes up approximately 76% of the total runtime and the grid update 24%. From this we can conclude that the performance of the algorithm is largely based on the performance of the FFT.

3.1 Accuracy

The NVIDIA Tesla supports only single precision floating points, which might decrease accuracy of the algorithm. To find out if the accuracy is problematic we have to look in more detail to the implementation of the IEEE 754 standard for floating points numbers.

[2]Remember that the Fourier transform has size $2(N+1)$ with N the number of points in the interior of the grid. Therefore a radix-2 FFT is possible when $N+1$ is a power of two ($N+1 = N_x - 1 = N_y - 1$).

A number is represented by a sign and *mantissa* or coefficient c scaled by a power of b,

$$(-1)^s \times c \times b^q.$$

The base $b = 2$ for most computer systems, the sign is either $s = 0$ or $s = 1$.
Four bytes are available to store the mantissa, the exponent and the position of the radix point. This limits the range of numbers that can be represented and the precision.

The gcc compiler presents several macros defining properties of the floating point representation. The most important ones for accuracy are listed in Table 3.1. The

FLT_RADIX	2
FLT_DIG	6
FLT_MANT_DIG	24
FLT_MIN	$1.175494 \cdot 10^{-38}$
FLT_EPSILON	$1.192093 \cdot 10^{-7}$

Table 3.1: Gcc macros for floating point numbers.

macro FLT_DIG defines the number of decimal digits of precision for the float type. In this case it is 6, which means that a float has 7 significant decimal digits. FLT_MIN is the smallest positive number that is representable in type float and FLT_EPSILON is the minimum positive floating point number such that the expression 1.0 + FLT_EPSILON != 1.0 yields true. FLT_MANT_DIG Is the number of digits in base-b that are used to store the mantissa. This implies that c can be at most $2^{24} - 1$.

With 7 significant decimal digits and a local truncation error of $\mathcal{O}\left(h^2\right)$ of the finite difference scheme we do not have to question the accuracy if the flow is not too large. For example on a 2049×2049 sized grid the truncation error is of order 10^{-6}. From Figure 3.3 we see that $\psi_{\max}$ is close to 4. This implies that the last decimal digit of the solution is probably not correct due to the numerical errors. However, if the flow is larger (100) the solution looses 'correct' digits. The same holds for computing the vorticity. Fortunately this is not the case in our simulations.

Other problems with floating point operations are the addition and subtraction of large and small numbers. This is because the result will be written in the base of the largest number and therefore digits might be lost. For example if we add

$$1450.563 + 3.141593 = 1453.705,$$

three digits of the smallest number are lost. To some extent this might be happening at the west boundary of the basin where $\frac{\partial \psi}{\partial x}$ is relatively large, but only for very small grid sizes.

Chapter 4

Conclusions and prospects

The speedup for this two-dimensional model is already impressive, but better performance can be expected for three dimensional models or for the double-layer barotropic vorticity model. When the problem size is increased, more threads can be swapped to hide idle time of threads. This also results in a larger speedup on smaller grids.

Three dimensional models are especially interesting, since they are suitable for employing multiple GPUs. The problem with multiple GPUs is that communication between them is needed and this involves moving data from one GPU to another via the CPU's memory. These memory operations are very slow compared to the bandwidth of global memory.

The main idea is to partition the domain and let each GPU do the computations of one sub-domain. Where two sub-domains meet, data is transferred between GPUs. The computational intensity for two dimensional models is too low to make it worthwhile to separate the data. For three dimensional models we can simply divide the domain in groups of adjoining horizontal layers.

GPUs can also be added to distributed memory systems to form a hybrid system. Implementations in, e.g., openMPI of ocean models can be accelerated using the GPU. The local computations then have to be ported to CUDA. However, the communication time is then increased by the latency of transferring data to the device memory.

The new Fermi architecture of NVIDIA's GPUs now has a 64 kB L1 cache per multiprocessor and a shared 768 kB L2 cache for global memory [9]. This improves bandwidth and reduces latency. Algorithms with low data parallelism, e.g. sparse matrix computations , can especially benefit from the L2 cache. Furthermore, the performance penalty of double precision is now a factor of 2 compared to a factor of 8 for devices

with compute capability 1.x. Another major improvement is the ability for concurrent kernel execution.

The GPU has proven itself as a viable alternative for, or addition to, the current HPC systems. The future of GPU computing is likely to be an exciting one where the boundaries between CPUs and GPUs are starting to disappear.

Bibliography

[1] Top500 supercomputing sites. `http://www.top500.org/lists/`.

[2] Pavel S. Berloff and James C. McWilliams. Large-scale, low-frequency variability in wind-driven ocean gyres. *Journal of Physical Oceanography*, 29(8):1925–1949, August 1999.

[3] J.C. Butcher. *Numerical Methods for Ordinary Differential Equations*, pages 98–99. John Wiley & Sons, Ltd, 2005.

[4] Márcia Alves de Inda. *Constructing Parallel Algorithms for Discrete Transforms*. PhD thesis, Universiteit Utrecht, 2000.

[5] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the ACM/IEEE SC—05 Conference*, November 2005.

[6] Mark Harris. Optimizing parallel reduction in cuda. `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`.

[7] NVIDIA Corporation. *CUDA C Best Practices Guide*, 3.2 edition, August 2010.

[8] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, 3.2 edition, September 2010.

[9] David Patterson. The top 10 innovations in the new nvidia fermi architecture, and the top 3 next challenges. `http://www.nvidia.com/content/PDF/fermi_white_papers/D.Patterson_Top10InnovationsInNVIDIAFermi.pdf`, September 2009.

[10] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, pages 510–514. Cambridge University Press, Cambridge, second edition, 1992.

[11] Markus Puschel and José M.F. Moura. Algebraic Signal Processing Theory: Cooley-Tukey Type Algorithms for DCTs and DSTs. *IEEE Transactions on Signal Processing*, 56(4):1502–1521, April 2008.

[12] G. Strang. *Computational Science and Engineering*, pages 283–288. Wellesley-Cambridge Press, 2007.

[13] Julien C. Thibault and Inanc Senocak. CUDA Implementation of a Navier-Stokes Solver on Multi-GPU Desktop Platforms for Incompressible Flows. In *47th AIAA Aerospace Sciences Meeting Including The New Horizons Forum and Aerospace Exposition*, 2009.

Appendix A

Hardware specifications

A.1 GPU specifications

The device specifications as displayed by the CUDA example program deviceQuery. This information was taken from `https://subtrac.sara.nl/userdoc/wiki/gpu/description`.

Device:	Tesla M1060
CUDA Capability Major revision number:	1
CUDA Capability Minor revision number:	3
Total amount of global memory:	4 Gbyte
Number of multiprocessors:	30
Number of cores:	240
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	16384
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	512 x 512 x 64
Maximum sizes of each dimension of a grid:	65535 x 65535 x 1
Maximum memory pitch:	262144 bytes
Texture alignment:	256 bytes
Clock rate:	1.30 GHz
Single Precision floating point performance (peak):	933 MFlop
Double Precision floating point performance (peak):	78 MFlop
Floating Point Precision:	IEEE 754 single & double
Memory speed:	800 MHz
Memory interface:	512 bit
Memory bandwidth:	102 Gbyte/sec
Concurrent copy and execution:	Yes
Run time limit on kernels:	No

Integrated:	No
Support host page-locked memory mapping:	Yes
Compute mode:	Default (multiple host threads can use this device simultaneously)

A.2 CPU specifications

One node from Lisa contains

- 2 quad CPU processors: Intel® Xeon® CPU L5420 @ 2.50GHz
- Front Side Bus: 1067 Mhz
- 32 GByte memory
- 233 GByte scratch disk
- 2 NVIDIA® Tesla graphic adapters

The Intel® Xeon® CPU L5420 has the following specifications:

Launch Date	Q1'08
Processor Number	L5420
# of Cores	4
# of Threads	4
Clock Speed	2.5 GHz
L2 Cache	12 MB
Bus/Core Ratio	7.5
FSB Speed	1333 MHz
FSB Parity	Yes
Instruction Set	64-bit
Lithography	45 nm
Max TDP	50 W
Tray 1ku Budgetary Price	$380.00

Table A.2: Source: `http://ark.intel.com/Product.aspx?id=33929`

Appendix B

Tables

B.1 Time per iteration

N	GPU ms/iter	CPU ms/iter
33	0.3235	0.2056
65	0.3338	0.7342
129	0.3883	3.147
257	0.7845	12.81
259	1.648	22.85
513	1.963	53.94
515	15.70	112.2
769	5.436	140.8
1025	6.646	260.5
2049	26.33	1266
4097	111.8	5337

Table B.1: Time per iteration in ms/iter

B.2 Speedup

N	Speedup
33	0.64
65	2.2
129	8.1
257	16.3
513	27.5
1025	39.2
2049	48.1
4097	47.7

Table B.2: Speedup