



UNIVERSITÉ CATHOLIQUE DE LOUVAIN
ÉCOLE POLYTECHNIQUE DE LOUVAIN
DÉPARTEMENT D'INGÉNIEURIE MATHÉMATIQUE

Discontinuous Galerkin Method on Graphics Processing Units

Implementation of a DGM for hyperbolic partial differential equations on GPUs

Master thesis presented by
Marc HENRY DE FRAHAN
for the degree of
Engineer in Applied Mathematics

Thesis committee:
J-F. REMACLE (Advisor)
P. CHATELAIN (Advisor)
V. LEGAT (Advisor)
J. LAMBRECHTS
R. KEUNINGS

Louvain-la-Neuve, June 2011

Je tiens à remercier toutes les personnes qui m'ont aidé lors de ce travail et qui m'ont accompagné et soutenu durant cette dernière année.

Je remercie particulièrement mes promoteurs Jean-François Remacle, Philippe Chatelain et Vincent Legat pour leur aide, leurs conseils et leur enthousiasme. Mon introduction au calcul scientifique hautement parallèle m'a beaucoup plu et enrichi. Leur encadrement et le projet m'a convaincu de continuer dans le domaine de la recherche et de poursuivre une thèse de doctorat à l'University of Michigan dans le domaine de la simulation numérique. Je remercie l'Unité de Mécanique Appliquée pour l'accès aux ordinateurs performants et aux GPUs.

Je tiens aussi à remercier Jonathan Lambrechts pour son aide tout le long de cette année. Ses réponses promptes à toutes mes questions allant de l'informatique et la connection aux machines MEMA à la méthode DG m'ont permis d'avancer rapidement dans mon travail.

Je veux aussi remercier ma famille pour leur soutien et mes co-koteurs pour leur bonne humeur et les scéances de basket-ball. Je remercie particulièrement Cassie Brown pour son soutien et la relecture de mon rapport. Je remercie enfin François-Xavier Mouthuy et Cyrille Dejemeppe pour leur conseils et leur aide lors de ma recherche de bugs informatiques.

Contents

Acknowledgments	iii
Introduction	1
1 Graphics Processing Units	3
1.1 The Device	3
1.2 Kernels	5
1.3 Thread Hierarchy	5
1.4 GPU Memory	7
2 The Discontinuous Galerkin Method	11
2.1 DGM formulation	12
2.2 Efficient assembly	14
2.2.1 Collocation	16
2.2.2 Evaluation	16
2.2.3 Redistribution	17
2.2.4 Face Contributions	18
2.2.5 Extension to a Non-Steady Conservation Law	19
3 The Shallow Water Equations	23
3.1 The Simplified Hyperbolic Partial Differential Equations	23
3.2 Tsunami Modeling	28
4 Implementing the Discontinuous Galerkin Method on the GPU	33
4.1 General Algorithms	33
4.2 Kernels and Parallel Implementation	35
5 Comparing CPU and GPU Implementations	39
5.1 Diagnostics	41
5.1.1 GPU Timers	41
5.1.2 Bandwidth	41
5.1.3 Floating Point Operations	43
5.1.4 Kernel occupancy	44

5.2	Preliminary Analysis	47
5.3	Individual Kernel Analysis	52
5.3.1	Equalizing Two Vectors	54
5.3.2	Mappings	54
5.3.3	Evaluating the Physics	56
5.3.4	Matrix-matrix Products	56
5.3.5	Solving the Linear System	57
5.4	Conclusion	58
6	GPU Code Improvements	61
6.1	High Priority Improvement Guidelines	61
6.2	Suggested Improvements to P_{GPU}	62
6.3	Alternative Discontinuous Galerkin Implementation	63
	Conclusion	65
	A Solution to the Benchmark Problem	69
	B Kernel Codes	71
	C Additional Figures and Tables	79
C.1	Launch Time Tables	79
C.2	Relative Standard Deviation Tables	80
C.3	Comparing Kernels	81
C.3.1	equal	82
C.3.2	Mappings	83
C.3.3	Evaluating the Physics	84
C.3.4	Multiplying by the Jacobians	86
C.3.5	Matrix-matrix Products	88
C.3.6	Solving the Linear System	92
D	Computer characteristics	93
D.1	CPU Characteristics	93
D.2	GPU Characteristics	93
E	Notations	97

Introduction

High performance computing is an important aspect of modern scientific simulations, and using computers to solve complex problems is central to modern science. Numerical simulation has become the third pillar of science, alongside theory and experimentation. Conventional high performance computing relies on supercomputers and large clusters of central processing units (CPU) working in parallel to solve a problem. With the advent of powerful graphics processing units (GPU) for gaming and visualization purposes, scientists have been looking to use these new machines for high performance computing.

A GPU is a computer chip specially designed to display images on a computer screen. Images are stored as vectors in computer memory. Manipulating them requires fast matrix and vector calculations. The GPU therefore is optimized to perform these operations efficiently, and it is designed to do intensive computation and massively parallel calculations. Its large number of processing cores (240 for the Tesla C1060) can handle thousands of threads concurrently. GPU threads are extremely lightweight and have little creation overhead, making the GPU ideal for high performance scientific computing. As with all parallel architectures, understanding the GPU structure is essential to use it efficiently. We present an introduction to GPU programming in Chapter 1.

Using the GPU's full potential requires an efficient parallel numerical method. Well suited to the GPU architecture, the Discontinuous Galerkin method (DGM) presented in Chapter 2 is a variation on the classical finite element method (Warburton *et al.*, 2009).

The hyperbolic differential equations solved in this dissertation, a simplified version of the 2D shallow water equations, are presented in Chapter 3. A more physical version of the partial differential equations is solved using the GPU to simulate a tsunami initiated in the Pacific Ocean off the coast of Japan.

Chapter 4 describes three different implementations of the DGM: (1) a CPU version which uses the BLAS library, (2) a GPU version with hand-coded kernels, (3) a GPU version using a combination of hand-coded kernels and the CUBLAS library, a BLAS library for GPUs.

The simplified shallow water equations are used in Chapter 5 as a bench-

mark to compare the three different DGM implementations. We show considerable speedup with the GPU implementations. For fifth order elements, the GPU hand-coded implementation is about fifteen times faster than the BLAS CPU version. With the help of CUBLAS, the GPU implementation is about fifty times faster than the CPU version. Some DGM steps, such as matrix-matrix products and matrix-matrix additions, can take full advantage of the parallel architecture and achieve considerably higher speedup than others.

In Chapter 6 we present future possible improvements and optimizations for the GPU codes. These are expected to increase code performance by taking advantage of different memory spaces and threading options.

Chapter 1

Graphics Processing Units

Graphics processing units are traditionally designed to manipulate graphics. They are optimized to perform fast, computationally intensive, and highly parallel vector operations. The GPU has a manycore multithreaded architecture, making it ideal for data parallel computation: the same program is executed on many different datasets. Our GPU, the Tesla C1060, has a theoretical bandwidth of 102.4 GB/s and a peak processing power of almost 1 TFLOPS in single precision calculation.

General-purpose computing on graphics processing units (GPGPU) used to be tedious. Intimate knowledge of the GPU architecture and programming language, both unsuitable for scientific computing, was assumed. Recently, GPU programming has been made accessible and programmer friendly through NVIDIA's Compute Unified Device Architecture (CUDA) programming model which includes a simple extension to the C programming language and the OpenCL framework.

Parallel systems continue to scale with Moore's Law. The CUDA programming model is designed to be very scalable. Parallel paradigms such as thread hierarchies, shared memory and barrier synchronizations are implemented, ensuring that current CUDA codes will scale well with future GPUs.

In addition, optimized libraries such as BLAS, LAPACK and FFT are being ported to GPUs. They are designed to take full advantage of its capabilities.

It is important to understand a GPU's architecture to program the DGM efficiently. We present an overview of the CUDA programming model: the device, the kernels, the thread hierarchies and the memory banks.

1.1 The Device

The GPU, which we call the device, is a set of multiprocessors and memory banks, schematically represented in Figure 1.1. The device contains N multiprocessors, 30 for the Tesla C1060. A multiprocessor contains several thread processors, eight for the Tesla C1060. Each processor has a Single

Instruction Multiple Data (SIMD) architecture: the processor executes the same instruction but on different datasets. All multiprocessors have access to a read-write device memory, a read-only constant cache, and a read-only texture cache. The host allocates device memory and sends and retrieves data from the global, constant, and texture memory. The processors on a multiprocessor have access to a shared memory which lies on the multiprocessor chip. Multiprocessors cannot access another multiprocessor's shared memory. Each processor has a set of read-write registers to store intermediate results. The multiprocessor instruction unit is responsible for the multiprocessor memory fetches and execution instructions.

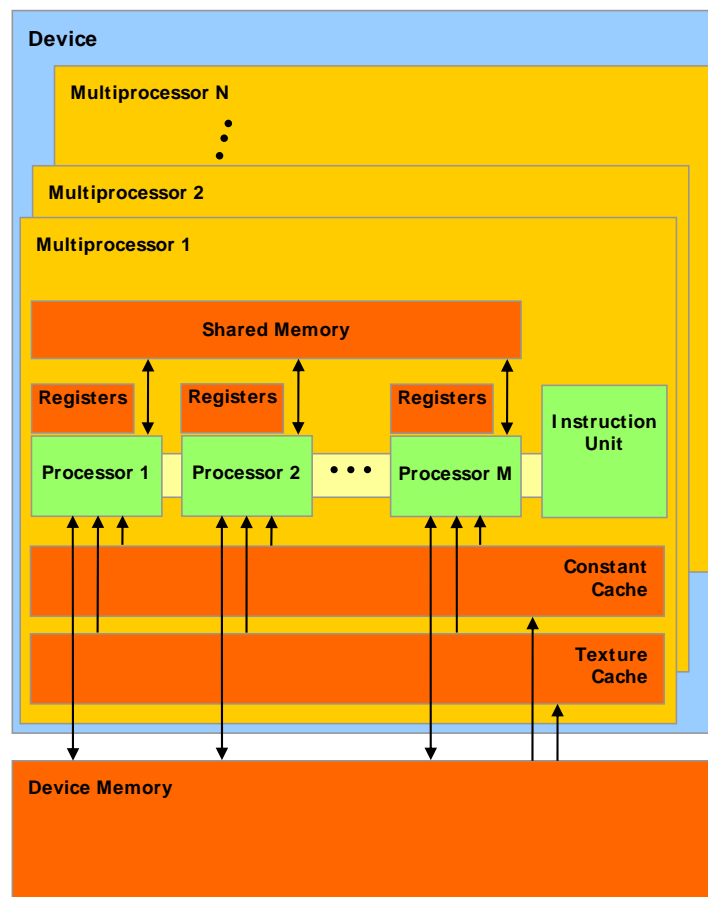


Figure 1.1: **Hardware model of the device.** The GPU, or device, has N multiprocessors and M processors per multiprocessor. Registers, shared memory, constant memory cache, and texture cache constitute the on-chip memory. Device memory is accessible by all threads. Source: NVIDIA Corporation (2008).

1.2 Kernels

Kernels are user defined C functions executed on the device. Kernels are called by the host (CPU) main thread. They are executed n times in parallel: n different CUDA threads are created, each handling one of these executions.

Kernels are launched asynchronously. Control is handed back to the host thread immediately after kernel launch and the host code keeps executing. The Tesla C1060 does not support concurrent kernel execution. This is supported by some devices with compute capability 2.0 and concurrent execution has to follow specific guidelines (section 3.2.7.3 of NVIDIA Corporation (2010)).

A typical host-device interaction is illustrated in Figure 1.2. Host sequential code is executed. The host thread asynchronously launches a kernel and keeps executing subsequent host code. Reaching another kernel launch, the host thread waits for the previous kernel to finish before launching the next one.

1.3 Thread Hierarchy

The GPU executes a kernel defined by a grid of blocks. These blocks are composed of threads.

For convenience, threads can be grouped into blocks to form one, two or three dimensional blocks enabling the user to tailor block sizes and dimensions according to his application. Each thread in a block is given a unique thread index which is accessed through the built-in variables `threadIdx`. Blocks can be organized into a one or two dimensional grid to form a grid of blocks (Figure 1.2). For our device, the maximum number of threads in a block is 512 and the maximum number of blocks is 65535.

A block resides on one multiprocessor. To hide memory latencies, a multiprocessor will handle several blocks concurrently. All threads on the multiprocessor share the resources of that multiprocessor. The device schedules the blocks so that each block has access to enough resources. However, the total memory needed by all the threads in one single block cannot exceed the multiprocessor resources.

Threads are executed concurrently. Threads within a block can communicate through the shared memory. Synchronization points can be used to synchronize threads within a block. Synchronizing is a lightweight operation. On the contrary, blocks are required to be independently executable; they have to be executable in any order. Blocks therefore can be scheduled on any number of cores, making the program highly scalable. This implies that threads from different blocks cannot communicate with each other and cannot be synchronized.

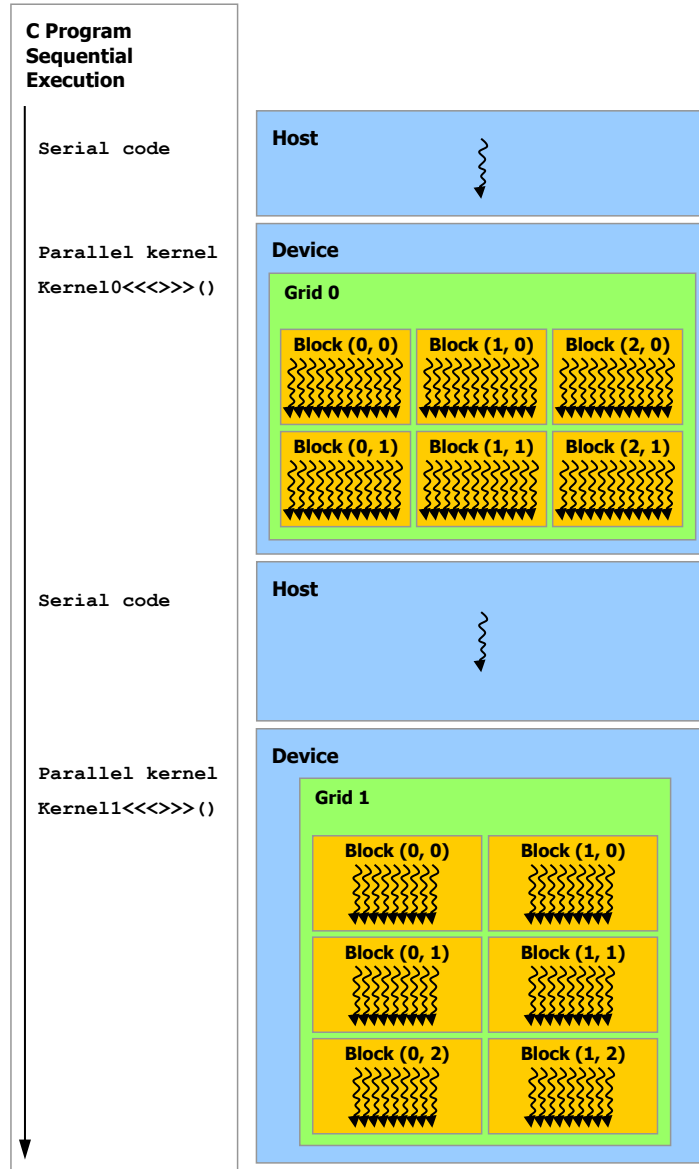


Figure 1.2: Execution flow of the main host thread and the device threads organized in a grid of blocks. The host thread executes serial code and launches the kernel. After a kernel launch the host code keeps executing until reaching another kernel launch. It will wait for the previous kernel to finish before launching the next kernel. A grid is a one or two dimensional structure of thread blocks. Each block is a one, two or three dimensional structure of individual threads. Source: NVIDIA Corporation (2010).

1.4 GPU Memory

CUDA threads have access to different levels of memory. We present a software model of the device memory in Figure 1.3. This figure should be contrasted with the hardware model of the device (Figure 1.1).

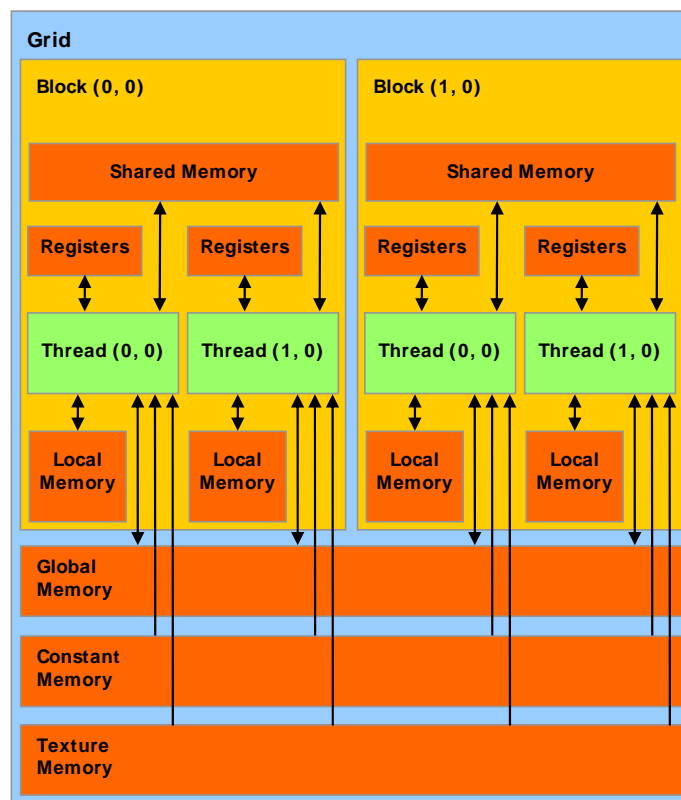


Figure 1.3: **Software memory model of the device.** Thread blocks have fast access to registers and shared memory. Threads in a block share the same shared memory. They cannot communicate with threads in other blocks. Each thread has its own registers. Off-chip memory such as global memory offers much higher memory latency and is accessible by any thread. Local memory also resides off-chip and suffers the same high latency as the global memory. Source: NVIDIA Corporation (2008).

- Each thread uses the **registers** of the multiprocessors. Registers are accessed per thread and have a very low latency of about 11 processor clock cycles. The number of registers per multiprocessor is fixed (16384 32-bits registers for the Tesla C1060), limiting the number of concurrent blocks on a multiprocessor.

- All threads of a same block access a very low latency **shared memory** (Figure 1.4). The amount of shared memory on a multiprocessor is fixed, limiting the number of concurrent blocks on the multiprocessor. This memory cannot be dynamically allocated by the threads.
- The **constant memory** and the **texture memory** are persistent on the device and can be read by all threads. The host writes on those device memories. The device threads access being read-only, these memories are cached on the multiprocessors, reducing the latency. The total amount of constant memory available for the Tesla C1060 is about 65 kB. Texture memory is accessed in several different ways, discussed in NVIDIA Corporation (2010).
- All threads access the device main memory, the **global memory**. This memory resides off-chip and has a very high latency of about 300 cycles. The total amount of device memory for the Tesla C1060 is about 4 GB. The **local memory** is a part of the device memory that acts as a spillover for registers. Variable arrays declared by a thread are automatically created in local memory unless otherwise specified by a declaration modifier.

We can synthesize these different memories in Table 1.1.

Memory	Penalty	Scope	Allocation	Size
register	1×	thread	static	16384 reg. per MP
local	100×	thread	dynamic	-
shared	1×	block	static	16384 B per MP
global	100×	global	dynamic	4 GB
constant	100× (cached)	global (ro)	n/a	65 kB

Table 1.1: Location, access and scope of the different memories. Access penalty for global memory is much higher than register or shared memory access. n/a = not applicable, ro = read-only, reg. = registers, MP = multiprocessor.

In the following chapter we discuss the numerical method that we implement on the GPU. The Discontinuous Galerkin method is a parallel method that can take full advantage of the GPU architecture.

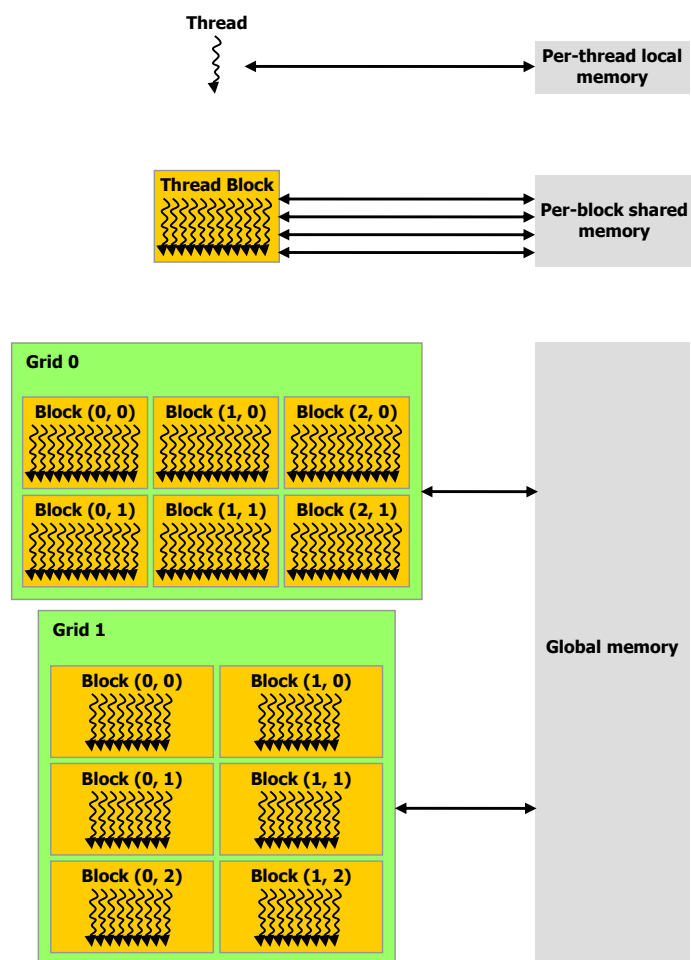


Figure 1.4: Memory hierarchy: each thread has its own memory, a block of threads share memory and all threads have access to global memory. Source: NVIDIA Corporation (2010).

Chapter 2

The Discontinuous Galerkin Method

The Discontinuous Galerkin method (DGM) is a discontinuous approach to the Finite Element method (FEM). The FEM, a numerical method used to find an approximated solution to a partial differential equation, presents many advantages. Interpolation theory ensures high order accuracy and convergence. It divides the problem domain into small pieces, called elements, which form the mesh. The FEM approximates the solution to the partial differential equation using these elements. The elements are flexible in size and shape and can capture complex boundaries and geometries. The DGM uses a Finite Volume method (FVM) approach to the flux terms, making it particularly suitable for hyperbolic partial differential equations and unstructured grids.

As opposed to classical FEM, adjacent elements in DGM do not share degrees of freedom on boundary nodes. Therefore, discrete fields can be discontinuous across the interfaces. These discontinuities are treated numerically by approximated Riemann solvers for the hyperbolic part of the equation.

There are two important advantages to the DGM approach. The DGM is naturally stable for advective problems, and it is a highly parallel algorithm because each element in the mesh is considered separate from the others. The latter advantage makes the DGM ideal for GPU implementation.

Unfortunately diffusion processes are expensive to evaluate and the DGM traditionally presents a higher computational cost than other methods. We address the latter issue in this dissertation.

In this chapter, following Lambrechts (2011), we present the DGM formulation and an efficient assembly of the DGM linear system. In Vos *et al.* (2010), a similar approach is used to achieve high efficiency in the assembly process through matrix-vector products. Lambrechts (2011) generalizes this approach to an efficient assembly using large matrix-matrix products.

2.1 DGM formulation

The problem of interest is to find the unknown $u \in V(\Omega)$ in a steady scalar conservation law

$$0 = \nabla \cdot \mathbf{f}(u, \nabla u) + s(u, \nabla u) \quad (2.1)$$

$$(2.2)$$

with $\Omega \subset R^D$, a function space with sufficient regularity $V(\Omega)$, the fluxes \mathbf{f} and the source term s . We discuss a scalar conservation law but extending the discussion to systems is trivial.

The problem domain Ω is meshed into a set of N_E non-overlapping elements, Ω_e ,

$$\Omega = \bigcup_{e=1}^{N_E} \Omega_e$$

We assume in this paper that all the elements belong to the same class of elements (triangular or quadrilateral or other types). This assumption is not necessary but it simplifies notations. Our discussion remains valid for mixed meshes. Each element is bounded by N_T faces: for example, a 2D triangular element has three interfaces.

The DGM consists of searching for the approximation u^h of the unknown u such that

$$u^h \in V_p^h = \{v \in L^2 : v|_{\Omega_e} \in P^p(\Omega), e = 1, \dots, N_E\}$$

with $P^p(\Omega)$ the space of polynomials of degree p on Ω_e and L^2 the space containing functions whose square can be integrated.

For each element we expand u in the following manner

$$u(\mathbf{x})|_{\Omega_e} \simeq u^h(\mathbf{x})|_{\Omega_e} = \sum_{i=1}^{N_s} U_{E(e,i)} \phi_{E(e,i)}(\mathbf{x}), \quad e = 1, \dots, N_E \quad (2.3)$$

with $U_{E(e,i)}$ the unknown nodal values on the element e , N_s the number of nodes on the element (also the number of nodal functions ϕ) and $E(e,i)$ a bi-jjective mapping from the physical element to the reference element, providing a global index of the degree of freedom ($E = 1, \dots, N_E N_s$, the number of nodes in the mesh) (Figure 2.1).

Multiplying Equation (2.1) by the test functions ϕ_I and then integrating over the domain leads to the standard DGM weak formulation

$$B(u, \phi_I) = \int_{\Omega} (\nabla \cdot \mathbf{f}(u, \nabla u)) \phi_I \, dv + \int_{\Omega} s(u, \nabla u) \phi_I \, dv = 0, \quad I = 1, \dots, N_E N_s$$

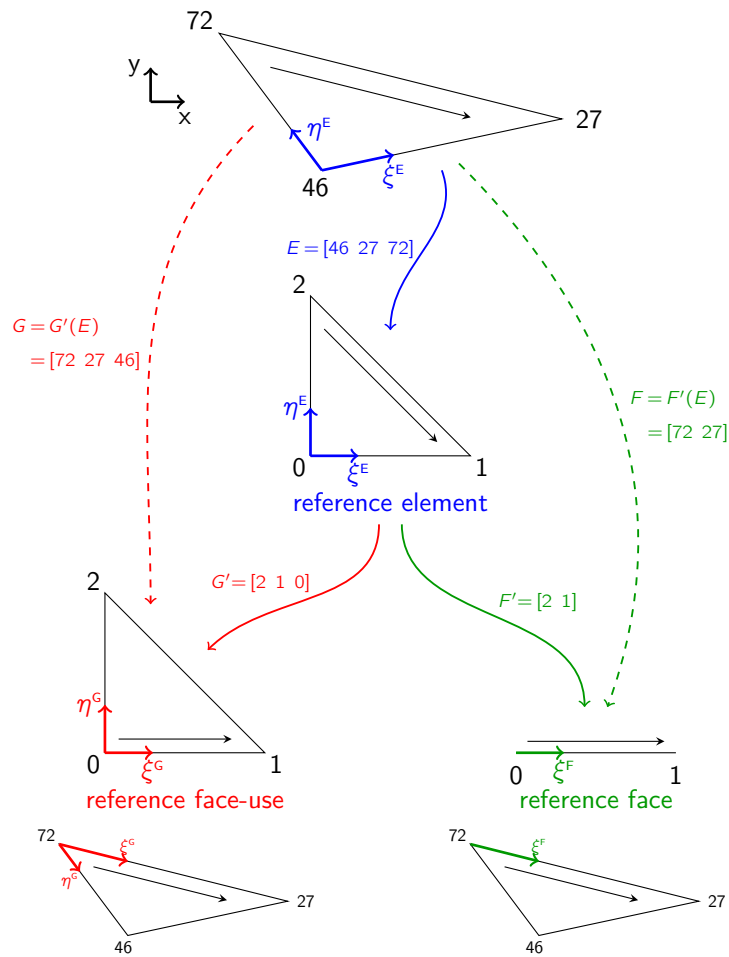


Figure 2.1: Element mapping between the physical element and the reference element, $E(e, i)$, and interface mapping between the physical interface and the reference interface, $F(t, c, j)$. Integration is done in the reference space. The mappings enable the transformation from the physical space to the reference space: **element integral**, **interface integral using all the nodal values** and **interface integral using the boundary nodes**. Source: Lambrechts (2011)

which is rewritten by integrating the divergence term by parts

$$B(u, \phi_I) = \int_{\Omega} (-\mathbf{f}(u, \nabla u)) \cdot \nabla \phi_I \, dv + \int_{\Gamma} q(u, \nabla u) \phi_I \, ds + \int_{\Omega} s(u, \nabla u) \phi_I \, dv = 0, \quad I = 1, \dots, N_E N_s \quad (2.4)$$

where $q = \mathbf{f} \cdot \mathbf{n}$ the normal flux is also called the numerical flux if Γ is an element interface or the boundary flux if Γ belongs to the domain boundary, $\partial\Gamma$.

The following three terms need to be evaluated

$$S_{ei} = \int_{\Omega_e} s(u, \nabla u) \phi_{E(e,i)} \, dv \quad (2.5)$$

$$F_{ei} = \int_{\Omega_e} -\mathbf{f}(u, \nabla u) \cdot \nabla \phi_{E(e,i)} \, dv \quad (2.6)$$

$$Q_{tcj} = \int_{\Gamma_t} q(u_d, \nabla u_d) \phi_{F(t,c,j)} \, ds \quad (2.7)$$

$\mathbf{S} = [S]_{ei}$ and $\mathbf{F} = [F]_{ei}$ are stored as matrices of size $N_s \times N_E$. There are therefore N_s equations to solve per element. Continuous and discontinuous formulation of \mathbf{S} and \mathbf{F} are identical. The expression for Q_{tcj} is valid for the DGM formulation only. For each interface in the mesh the normal fluxes are evaluated on each side of the edge ($c = 0, 1$). $F(t, c, j)$ is a mapping from the global edge nodes to the reference edge associated with node j of side c of face t (Figure 2.1). We note the number of nodes on the interface M_s and the number of interfaces in the mesh M_T .

The objective is to evaluate \mathbf{S} , \mathbf{F} and Q_{tcj} efficiently. We present an outline of the assembly procedure, discussed in depth in (Lambrechts, 2011).

2.2 Efficient assembly

Integration of Equations (2.5) and (2.6) over the element's region Ω_e is expressed as the integration over the reference element's region ω in the reference space ξ . Coordinates in the physical space are $x_a, a = 1, \dots, D'$ and coordinates in the reference space are $\xi_\alpha, \alpha = 1, \dots, D$ (Figure 2.1). Therefore, we can rewrite Equation (2.5)

$$S_{ei} = \int_{\Omega_e} s(u, \nabla u) \phi_{E(e,i)} \, dv = \int_{\omega} s(u, \nabla u) \phi_{E(e,i)} J \, dv$$

where J the determinant of the Jacobian matrix of the transformation $\mathbf{x}(\xi^E)$ from ω to Ω_e . For a given element e , the Jacobian in 2D is

$$Jac = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix}$$

and $J = \left| \frac{\partial x}{\partial \xi} \frac{\partial y}{\partial \eta} - \frac{\partial y}{\partial \xi} \frac{\partial x}{\partial \eta} \right|$

The Jacobian matrix and its determinant are precalculated and stored for all the elements.

We use a Gauss quadrature rule to approximate the integral by a sum over N_G points $\mathbf{x}_e(\Xi_g^E)$ on the element Ω_e where Ξ_g^E are the coordinates of the integration points in the reference space.

$$S_{ei} \simeq \sum_g^{N_G} [s\phi_{E(e,i)}J]_{\mathbf{x}_{eg}} w_g, \quad \forall e, i$$

where w_g is the weight of the g integration point and the functions enclosed in square brackets are evaluated at \mathbf{x}_{eg} . A similar procedure for the flux terms, Equation (2.6), defines

$$\begin{aligned} F_{ei} &= \int_{\Omega_e} -\mathbf{f}(u, \nabla u) \cdot \nabla \phi_{E(e,i)} \, dv \\ &= \int_{\Omega_e} \sum_a^D f_a(u, \nabla u) \frac{\partial \phi_{E(e,i)}}{\partial x_a} \, dv \\ &= \int_{\omega} \sum_a^D f_a(u, \nabla u) \sum_{\alpha}^{D'} \frac{\partial \phi_{E(e,i)}}{\partial \xi_{\alpha}^E} \frac{\partial \xi_{\alpha}^E}{\partial x_a} J \, dv' \end{aligned}$$

where $\frac{\partial \xi_{\alpha}^E}{\partial x_a}$ are the entries of the inverse Jacobian matrix and are precalculated for all the elements.

Applying the Gauss quadrature rule leads to an explicit formulation for \mathbf{F} .

$$F_{ei} \simeq \sum_g^{N_G} \sum_a^D \sum_{\alpha}^{D'} \left[f_a \frac{\partial \phi_{E(e,i)}}{\partial \xi_{\alpha}^E} \frac{\partial \xi_{\alpha}^E}{\partial x_a} J \right] w_g \quad \forall e, i$$

We will now show how to evaluate \mathbf{S} and \mathbf{F} efficiently in three steps:

1. Collocation: evaluating the unknowns at the integration points.
2. Evaluation: evaluating the physics at the integration points.
3. Redistribution: redistributing the flux and source terms to the element nodes.

The equivalent expressions for Q_{tcj} will be shown in Section 2.2.4.

2.2.1 Collocation

The Gauss quadrature formula requires that the unknowns and their gradients be evaluated at the element's integration points. We assume the elements belong to the same class, with the same nodal functions ϕ and integration points. The element's nodal functions and their gradients are

$$\begin{aligned} [\phi_{E(e,i)}]_{\mathbf{x}_{eg}} &= [\phi_i]_{\Xi_g^E} & \forall e, i, g \\ \left[\frac{\partial \phi_{E(e,i)}}{\partial \xi_\alpha^E} \right]_{\mathbf{x}_{eg}} &= \left[\frac{\partial \phi_i}{\partial \xi_\alpha^E} \right]_{\Xi_g^E} & \forall e, i, g \end{aligned}$$

The collocation step is written in the following way

$$\underbrace{[u]_{\mathbf{x}_{eg}}}_{A_0[g][e]} = \sum_i^{N_s} \underbrace{[\phi_i]_{\Xi_g^E}}_{w_0[g][i]} \underbrace{U_{E(e,i)}}_{X_0[i][e]} \quad \forall e, g \quad N_E N_G N_s \quad (2.8)$$

$$\underbrace{\left[\frac{\partial u}{\partial \xi_\alpha^E} \right]_{\mathbf{x}_{eg}}}_{A_1[g\alpha][e]} = \sum_i^{N_s} \underbrace{\left[\frac{\partial \phi_i}{\partial \xi_\alpha^E} \right]_{\Xi_g^E}}_{W_1[g\alpha][i]} \underbrace{U_{E(e,i)}}_{X_0[i][e]} \quad \forall e, g \quad N_E N_G N_s D' \quad (2.9)$$

where the right column indicates the number of operations for the sum. $A_1[g\alpha][e]$ is a $N_G D' \times N_E$ matrix with compressed row index $[gD' + \alpha]$ and column index $[e]$. Each integration point requires a sum of N_s shape functions. The collocation step therefore scales as $N_G N_s$. Both these numbers are proportional and scale like p^D , where p is the interpolation polynomial degree. It is essential to realize that these sums can be expressed as matrix-matrix products

$$\begin{aligned} \mathbf{A}_0 &= \mathbf{W}_0 \cdot \mathbf{X}_0 \\ \mathbf{A}_1 &= \mathbf{W}_1 \cdot \mathbf{X}_0 \end{aligned}$$

where \mathbf{W}_0 and \mathbf{W}_1 are constant and can be precalculated for all elements. This matrix-matrix product can be efficiently done on a computer with the Basic Linear Algebra Subprograms (BLAS level 3 library). The product can easily be extended to the case of systems of partial differential equations (Figure 2.2).

2.2.2 Evaluation

The evaluation step consists in calculating the physics of the problem.

The gradient of the unknowns in the physical space are evaluated with the gradients in the reference space calculated in the collocation step, Equation (2.9), and the inverse Jacobian matrix

$$\left[\frac{\partial u}{\partial x_a} \right]_{\mathbf{x}_{eg}} = \sum_\alpha^{D'} \left[\frac{\partial u}{\partial \xi_\alpha^E} \right]_{\mathbf{x}_{eg}} \left[\frac{\partial \xi_\alpha^E}{\partial x_a} \right]_{\mathbf{x}_{eg}} \quad \forall e, g, a \quad N_E N_G D D' \quad (2.10)$$

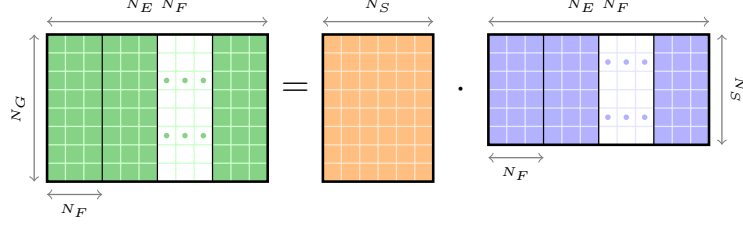


Figure 2.2: Evaluating the unknowns at the integration points with the collocation matrix-matrix products, Equations (2.8) and (2.9), extended to a system of partial differential equations containing N_F unknown fields. The three matrices are \mathbf{A}_0 or \mathbf{A}_1 , \mathbf{W}_0 or \mathbf{W}_1 and \mathbf{X}_0 . Source: Lambrechts (2011).

Computing this expression is not as efficient as the collocation step (it cannot be written as a matrix-matrix product) but it requires fewer operations. The ratio of the number of operations for evaluating Equation (2.9) to evaluating Equation (2.10) is $\frac{N_s}{D}$, which is typically much greater than one. Evaluating Equation (2.9) is therefore more computationally expensive.

The physics of the problem is evaluated as

$$[s]_{\mathbf{x}_{eg}} = s \left([u]_{\mathbf{x}_{eg}}, \left[\frac{\partial u}{\partial x_a} \right]_{\mathbf{x}_{eg}}, \mathbf{x}_{eg} \right) \quad \forall e, g \quad \mathcal{O}(N_E N_G) \quad (2.11)$$

$$[f_a]_{\mathbf{x}_{eg}} = f_a \left([u]_{\mathbf{x}_{eg}}, \left[\frac{\partial u}{\partial x_a} \right]_{\mathbf{x}_{eg}}, \mathbf{x}_{eg} \right) \quad \forall e, g, a \quad \mathcal{O}(N_E N_G D) \quad (2.12)$$

The physics simulated determines the flux and source functions, which imply that complicated physics will lead to a more expensive evaluation step. However, the number of operations scale with N_G and not $N_G N_s$. For high order elements the cost of evaluating the physics is typically much smaller than the cost of collocation.

2.2.3 Redistribution

The redistribution step takes the geometry into consideration. Local contributions calculated at the evaluation step are first multiplied by the element's Jacobian matrix at the integration points, $[J]_{\mathbf{x}_{eg}}$.

$$[sJ]_{\mathbf{x}_{eg}} = [s]_{\mathbf{x}_{eg}} [J]_{\mathbf{x}_{eg}} \quad \forall e, g \quad N_E N_G \quad (2.13)$$

$$[f'_\alpha J]_{\mathbf{x}_{eg}} = \sum_a^D \left[\frac{\partial \xi_\alpha^E}{\partial x_a} \right]_{\mathbf{x}_{eg}} [f_a]_{\mathbf{x}_{eg}} [J]_{\mathbf{x}_{eg}} \quad \forall e, g, \alpha \quad N_E N_G D' D \quad (2.14)$$

Because the number of operations goes as N_G and not $N_G N_s$, these operations are relatively cheap.

The objective of the redistribution step is to redistribute the contributions evaluated at the integration points to the element nodes, using the Gauss quadrature integration formula

$$\underbrace{S_{ei}}_{A_2[i][e]} \simeq \sum_g^{N_G} \underbrace{[\phi_i]_{\Xi_g^E}}_{W_2[i][g]} \underbrace{[sJ]_{\mathbf{x}_{eg}}}_{X_2[g][e]} \quad \forall e, i \quad N_E N_s N_G \quad (2.15)$$

$$\underbrace{F_{ei}}_{A_3[i][e]} \simeq \sum_g^{N_G} \sum_\alpha^{D'} \underbrace{\left[-\frac{\partial \phi_i}{\partial \xi_\alpha^E} \right]_{\Xi_g^E}}_{W_3[i][g\alpha]} w_g \underbrace{[f'_\alpha J]_{\mathbf{x}_{eg}}}_{X_3[g\alpha][e]} \quad \forall e, i \quad N_E N_s N_G D' \quad (2.16)$$

The number of operations scale like $N_G N_s$, making this step as expensive as collocation. However, redistribution can be expressed as a matrix-matrix operation as well and benefit from the same BLAS advantages.

$$\mathbf{S} = \mathbf{W}_2 \cdot \mathbf{X}_2$$

$$\mathbf{F} = \mathbf{W}_3 \cdot \mathbf{X}_3$$

We have divided the assembly of \mathbf{S} and \mathbf{F} in three steps. The first and third step are expensive but can be expressed as matrix-matrix products and computed efficiently. Evaluating the physics is less costly but cannot be done efficiently.

2.2.4 Face Contributions

We now apply these steps to assemble the third matrix, \mathbf{Q} , representing the contributions from the fluxes through the interfaces.

The objective is to evaluate the integral

$$Q_{tcj} = \int_{\Gamma_t} q_c(u_d, \nabla u_d) \psi_{F(t,c,j)} ds$$

where ψ_j are the nodal functions of the reference interface and $F(t, d, j)$ maps the degree of freedom on a physical interface to a node on a reference face defined in the parametric space ξ (Figure 2.1).

The collocation step is similar to the previous one.

$$\underbrace{[u_{td}]_{\mathbf{x}_{tg}}}_{A_4[g][td]} = \sum_j^{M_s} \underbrace{[\psi_j]_{\Xi_g^F}}_{W_4[g][j]} \underbrace{U_{F(t,d,j)}}_{X_4[j][td]} \quad \forall t, d, g \quad M_T M_G 2 M_S \quad (2.17)$$

The number of operations is defined by the product of the number of interfaces, M_T , the number of integration points on the interface, M_G , the

number of sides of each interface, indexed by $d = (0, 1)$, and the number of nodes on an interface, M_s . This operation is computed efficiently with a matrix-matrix product

$$\mathbf{A}_4 = \mathbf{W}_4 \cdot \mathbf{X}_4$$

We will not be discussing diffusion processes in this work so we neglect the gradients of the face. Their evaluation can be found in Lambrechts (2011).

Evaluating the physics cannot be done efficiently though the operations scale with M_G and not $M_G M_s$. It is therefore less costly than the collocation step.

$$[q_c]_{\mathbf{x}_{tg}} = q_c \left([u_d]_{\mathbf{x}_{tg}}, \mathbf{x}_{tg} \right) \quad \forall t, c, g \quad \mathcal{O}(M_T M_G^2) \quad (2.18)$$

Multiplying by the face Jacobians takes into account the geometry.

$$[q_c J]_{\mathbf{x}_{tg}} = [q_c]_{\mathbf{x}_{tg}} [J]_{\mathbf{x}_{tg}} \quad \forall t, g, c \quad \mathcal{O}(M_T M_G^2) \quad (2.19)$$

Finally the redistribution step can be written as a matrix-matrix product.

$$\underbrace{Q_{tcj}}_{A_5[j][tc]} \simeq \sum_g^{M_G} \underbrace{[\psi_j]_{\Xi_g^F} w_g}_{W_5[j][g]} \underbrace{[q_c J]_{\mathbf{x}_{tg}}}_{X_5[g][tc]} \quad \forall t, c, j \quad M_T M_s M_G^2 \quad (2.20)$$

There are two implicit steps that have to be added to the assembling of the face terms. Before the collocation step the interface unknowns $U_{F(t,d,j)}$, stored in \mathbf{X}_4 , have to be mapped from the element unknowns $U_{E(e,i)}$, stored in \mathbf{X}_0 . This mapping from the element to the interfaces is necessary for the efficient calculation of the face contributions. Once Q_{tcj} is assembled, these contributions have to be mapped back to the element nodes to construct Q_{ei} .

Putting the source, flux and face contributions together we rewrite the weak formulation, Equation (2.4), in terms of the following matrices, each of size $N_s \times N_E$,

$$\mathbf{B} = \mathbf{S} + \mathbf{F} + \mathbf{Q} = 0$$

2.2.5 Extension to a Non-Steady Conservation Law

We extend our results to a non-steady scalar conservation law:

$$\frac{\partial u}{\partial t} = \nabla \cdot \mathbf{f}(u, \nabla u, t) + s(u, \nabla u, t) \quad (2.21)$$

$$(2.22)$$

We now study the left hand side as the evaluation of the right hand term has already been done. Weak formulation of the time derivative is expressed

$$\int_{\Omega} \frac{\partial u}{\partial t} \phi_I \, dv = \frac{\partial}{\partial t} \int_{\Omega} u \phi_I \, dv \quad \forall I = 1, \dots, N_E N_s$$

because the nodal functions and element integrals are time independent.

Expanding u with Equation (2.3), we compute the mass matrix for each element

$$\begin{aligned} \int_{\Omega_e} u \phi_j \, dv &\simeq \int_{\Omega_e} \sum_{i=1}^{N_s} U_{E(e,i)} \phi_{E(e,i)} \phi_{E(e,j)} \, dv && \forall e \quad i, j = 1, \dots, N_s \\ &= \sum_{i=1}^{N_s} \int_{\Omega_e} \phi_{E(e,i)} \phi_{E(e,j)} \, dv U_{E(e,i)} \\ &= \mathbf{M}_e U_{E(e,i)} \end{aligned}$$

The entries of the mass matrix \mathbf{M}_e are

$$[M_e]_{ij} = \int_{\Omega_e} \phi_{E(e,i)} \phi_{E(e,j)} \, dv \quad \forall e \quad i, j = 1, \dots, N_s$$

The integral is approximated with a Gauss quadrature.

$$[M_e]_{ij} \simeq \sum_g^{N_G} [\phi_{E(e,i)} \phi_{E(e,j)} J]_{\mathbf{x}_{eg}} w_g \quad \forall e \quad i, j = 1, \dots, N_s$$

Each mass matrix is symmetric and constant for each element. They are calculated once for all elements in the mesh.

The non-steady conservation law, Equation (2.21), is written for each element

$$\begin{aligned} \mathbf{M}_e \frac{\partial}{\partial t} \mathbf{U}_e(t) &= \mathbf{S}_e(t) + \mathbf{F}_e(t) + \mathbf{Q}_e(t) && \forall e \\ \Leftrightarrow \frac{\partial}{\partial t} \mathbf{U}_e(t) &= \mathbf{M}_e^{-1} (\mathbf{S}_e(t) + \mathbf{F}_e(t) + \mathbf{Q}_e(t)) && \forall e \end{aligned} \quad (2.23)$$

where $\mathbf{S}_e(t)$ is the column of \mathbf{S} associated with the element e . The inverse mass matrices \mathbf{M}_e^{-1} are constant and can be precalculated for each element. Time integration of the linear system can be done with a traditional time integration scheme such as the Runge-Kutta fourth order method. Because \mathbf{M}_e^{-1} is precalculated solving the linear system at each time integration step is just a matrix-vector product. \mathbf{S} , \mathbf{F} and \mathbf{Q} are evaluated in three steps: collocation, evaluation and redistribution. The first and third steps can be done efficiently. The second step is less efficient but less costly.

It is important to emphasize the parallel numerical method we have presented. The source and flux contributions are done by treating the elements

independently of each other. The face contributions can be evaluated by also treating each interface separately. Communication between the elements is done by gathering these contributions together and solving the linear system.

In Chapter 4 we present our parallel implementation of the DGM.

Chapter 3

The Shallow Water Equations

3.1 The Simplified Hyperbolic Partial Differential Equations

As stated in the introduction, our objective is to solve hyperbolic partial differential equations using the DGM on a GPU. We have chosen to solve the simplified 2D shallow water equations. These equations are obtained by depth-integrating the Navier-Stokes equations in the case where the horizontal length scale is much greater than the vertical length scale. These equations are used to model shallow fluid flows such as rivers and dam breaks. With the addition of the appropriate physics such as the Coriolis force, dissipation and friction forces, the equations can be used to model oceanic and atmospheric flows. In Section 3.2 we present the simplified equations governing a tsunami and the simulation of a tsunami off the coast of Japan.

The linear equations solved with the DGM are

$$\begin{cases} \frac{\partial \eta}{\partial t} = -h_0 \nabla \cdot \mathbf{u} \\ \frac{\partial \mathbf{u}}{\partial t} = -g_0 \nabla \eta \end{cases} \quad (3.1)$$

$$\Leftrightarrow \frac{\partial}{\partial t} \begin{bmatrix} \eta \\ u_x \\ u_y \end{bmatrix} = -\frac{\partial}{\partial x} \begin{bmatrix} h_0 u_x \\ g_0 \eta \\ 0 \end{bmatrix} - \frac{\partial}{\partial y} \begin{bmatrix} h_0 u_y \\ 0 \\ g_0 \eta \end{bmatrix} \quad (3.2)$$

where η is the relative (to a reference level) water elevation and \mathbf{u} is the depth-average mean velocity. The number of unknown fields, N_F is equal to three. The bathymetry is considered constant and equal to h_0 . Gravity is constant and equal to g_0 . A simple 1D representation of the variables and parameters is presented in Figure 3.1.

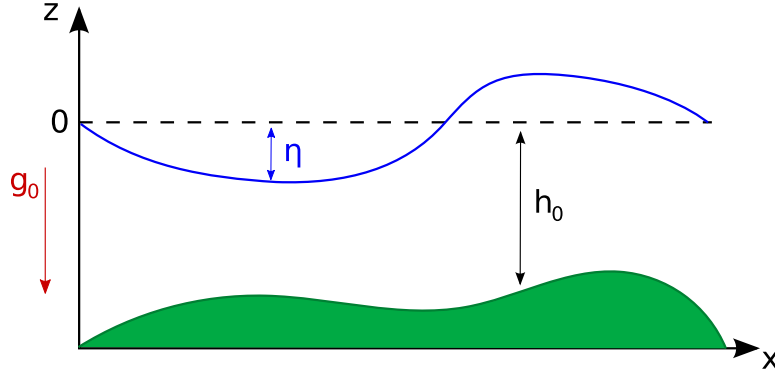


Figure 3.1: 1D illustration of the relative water elevation, η , and a non-constant bathymetry, $h_0(x)$.

Equation (3.1) defines a 2D wave equation. Using the non-dimensional variables

$$\eta' = \frac{\eta}{h_0} \quad \text{and} \quad \mathbf{u}' = \sqrt{\frac{g_0}{h_0}} \mathbf{u}$$

Equation (3.1) is rewritten

$$h_0 \frac{\partial \eta'}{\partial t} = -h_0 \sqrt{\frac{h_0}{g_0}} \nabla \cdot \mathbf{u}' \quad (3.3)$$

$$\sqrt{\frac{h_0}{g_0}} \frac{\partial \mathbf{u}'}{\partial t} = -g_0 h_0 \nabla \eta' \quad (3.4)$$

which is rewritten by deriving Equation (3.3) with respect to time and substituting with Equation (3.4)

$$\begin{aligned} \frac{\partial^2 \eta'}{\partial t^2} &= -\nabla \cdot \frac{\partial}{\partial t} \left(\sqrt{\frac{h_0}{g_0}} \mathbf{u}' \right) \\ &= -\nabla \cdot (-g_0 h_0 \nabla \eta') \\ &= g_0 h_0 \nabla^2 \eta' \\ &= c^2 \nabla^2 \eta' \end{aligned} \quad (3.5)$$

where the constant $c = \sqrt{g_0 h_0}$ is the wave propagation speed.

We present here a simple 2D simulation. To create an initial elevated bump of water, the initial conditions are

$$\begin{aligned} \eta(x, y, t = 0) &= \frac{1}{2} \exp(-20(x + 0.4)^2 - 20(y + 0.4)^2) \\ u_x(x, y, t = 0) &= 0 \\ u_y(x, y, t = 0) &= 0 \end{aligned}$$

and are illustrated in Figure 3.2a.

To simulate impermeable boundaries so that the fluid is completely enclosed, the boundary conditions are

$$\frac{\partial \eta}{\partial x}(\partial \Gamma, t) = 0 \tag{3.6}$$

$$\mathbf{u} \cdot \mathbf{n}(\partial \Gamma, t) = 0 \tag{3.7}$$

where \mathbf{n} is the normal to the boundary. Physically the model simulates an initial elevated bump of water in a enclosed square bassin ($L = 2$). Figure 3.2 shows the time evolution of this bump of water when $h_0 = 0.1$ m and $g_0 = 10$ m/s². The initial height of the bump decreases as a function of time, converting potential energy into kinetic energy (Figure 3.2d). The down rushing water pushes the adjacent water up the sides of the containment vessel (Figure 3.2j). This simulation was done using our GPU implementation of the DGM method.

The benchmark test in this report uses $h_0 = 1$ m, $g_0 = 1$ m/s² and the initial conditions

$$\eta(x, y, t = 0) = \exp(-2x^2 - 2y^2)$$

$$u_x(x, y, t = 0) = 0$$

$$u_y(x, y, t = 0) = 0$$

Using this problem, we will compare the different CPU and GPU implementations. The time evolution of this problem is presented in Appendix A.

To solve the shallow water equations using the DGM, it is useful to rewrite Equation (3.2) in the form of the non-steady conservation law presented in Equation (2.21). We identify the following terms

$$\mathbf{u} = \begin{bmatrix} \eta \\ u_x \\ u_y \end{bmatrix}$$

$$\mathbf{s} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{f} = \begin{bmatrix} -h_0 u_x & -h_0 u_y \\ -g_0 \eta & 0 \\ 0 & -g_0 \eta \end{bmatrix}$$

The flux term \mathbf{q} is given by

$$\mathbf{q} = \mathbf{f} \cdot \mathbf{n} = \begin{bmatrix} -h_0 u'_n \\ -g_0 \eta'_n \\ -g_0 \eta'_n \end{bmatrix}$$

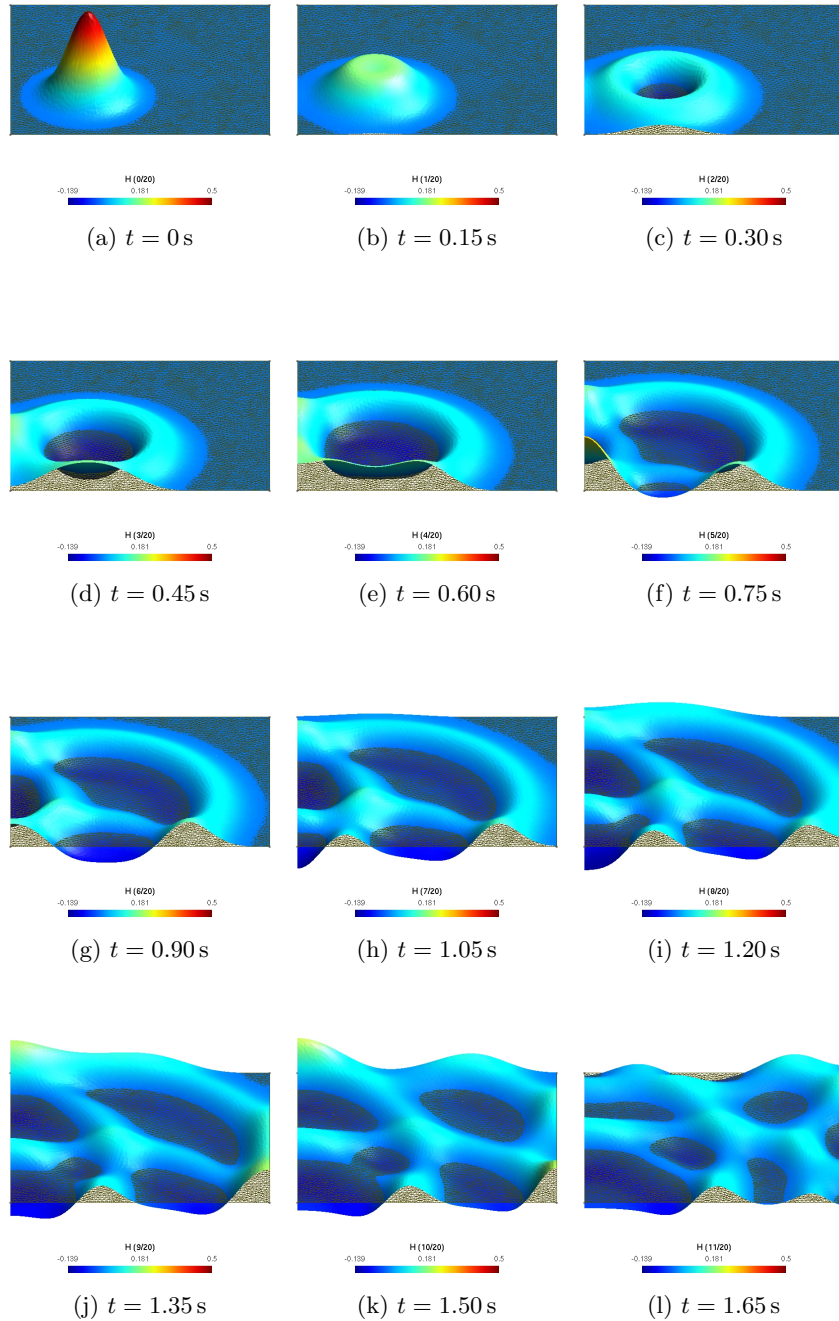


Figure 3.2: Relative water elevation, η , as a function of time with $h_0 = 0.1$ m and $g_0 = 10$ m/s². This simulation was done using our GPU implementation of the DGM method.

3.1 The Simplified Hyperbolic Partial Differential Equations 27

with $\mathbf{n} = (n_x, n_y)$ is the interface normal and where the appropriate Riemann solver defines

$$u'_n = \bar{u}_n + \sqrt{\frac{g_0}{h_0}} [u_n]$$

$$\eta' = \bar{\eta} + \sqrt{\frac{h_0}{g_0}} [\eta]$$

with the operators

$$\bar{a} = \frac{a_L + a_R}{2} \quad \text{the mean value of } a \text{ over the boundary}$$

$$[a] = \frac{a_L - a_R}{2} \quad \text{the jump of } a \text{ through the boundary}$$

and the indexes L and R denote the value of a on the left and right of the boundary, respectively. For a given face we can then write the normal fluxes in the following way

$$\mathbf{q}_L = \begin{bmatrix} -\frac{1}{2}h_0 \left(u_{Ln} + u_{Rn} + \sqrt{\frac{g_0}{h_0}}(\eta_L - \eta_R) \right) \\ -\frac{1}{2}g_0n_x \left(\eta_L + \eta_R + \sqrt{\frac{h_0}{g_0}}(u_{Ln} - u_{Rn}) \right) \\ -\frac{1}{2}g_0n_y \left(\eta_L + \eta_R + \sqrt{\frac{h_0}{g_0}}(u_{Ln} - u_{Rn}) \right) \end{bmatrix} = -\mathbf{q}_R$$

These expressions are developed and explained fully in Bernard (2008).

3.2 Tsunami Modeling

To simulate a tsunami initiated in the Pacific Ocean off the coast of Japan, we include additional physics in the shallow water equations (Equation (3.2)). The equations are solved on a stereographic projection of the Earth (Figure 3.3b) meaning that the problem is solved in 2D. The derivation of these equations in the stereographic coordinates are presented in a working paper by Slaoui (2011).

The equations describing the tsunami are the following

$$\begin{aligned}\frac{\partial \eta}{\partial t} &= -h_0(x, y) \frac{\partial u_x}{\partial x} - h_0(x, y) \frac{\partial u_y}{\partial y} \\ \frac{\partial u_x}{\partial t} &= -g_0 \frac{\partial \eta}{\partial x} + f u_y - c_D u_x - \frac{x}{2R^2} g_0 \eta \\ \frac{\partial u_y}{\partial t} &= -g_0 \frac{\partial \eta}{\partial y} - f u_x - c_D u_y - \frac{y}{2R^2} g_0 \eta\end{aligned}$$

where $h_0(x, y)$ is the non-constant depth of the ocean (Figure 3.4), R is the radius of the earth, x and y are the stereographic coordinates. We have added the Coriolis force term, $f u_y$ and $-f u_x$, where

$$f = 2\omega \sin(\theta)$$

with θ the latitude

The drag on the bottom of the ocean, $-c_D u_x$ and $-c_D u_y$, where

$$c_D = g_0 0.025^2 (\eta(x, y) + h_0(x, y))^{-\frac{4}{3}}$$

The stereographic terms, $-\frac{x}{2R^2} g_0 \eta$ and $-\frac{y}{2R^2} g_0 \eta$, come from the change of coordinates from a 3D sphere to a 2D projection of the Earth (Slaoui (2011)). Details about this mathematical model are found in Lambrechts (2011) and Bernard (2008).

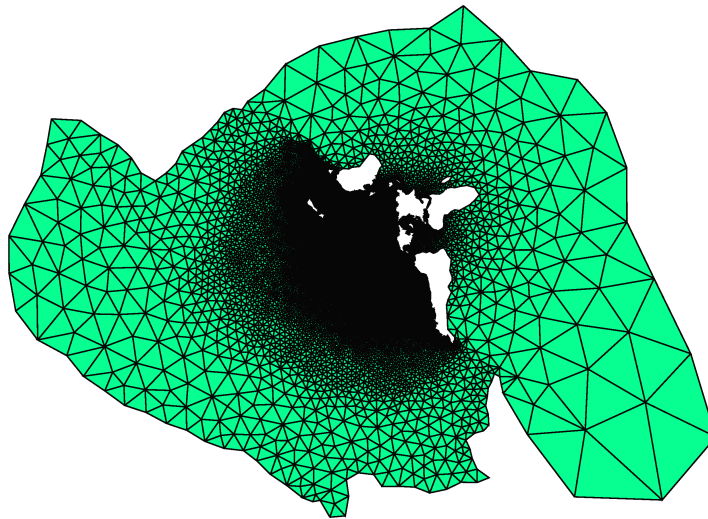
The initial condition for the initial water elevation is presented in Figure 3.5. Mathematical models of earthquake induced water elevation deformations are used to calculate this initial condition. The initial condition, taken from the study of Yushiro Fujii and Kenji Satake¹, is based on models for the 2011 Tohoku earthquake that occurred off the coast of Japan on March 11, 2011. The time evolution of the modeled tsunami is presented in Figure 3.6. The wave is initiated by an earthquake off the coast of Japan (Figure 3.6a). The waves propagate across the ocean and interact with bathymetric features and continental coasts (Figure 3.6f).

Successful use of the DGM to solve hydrodynamics problems and to model oceans flows has been shown in Dawson & Proft (2004), Kubatko *et al.* (2006) and Remacle *et al.* (2006).

¹<http://iisee.kenken.go.jp/staff/fujii/OffTohokuPacific2011/tsunami.html>



(a) Mesh of the Earth refined in the Pacific Ocean.



(b) Stereographic projection of the Earth.

Figure 3.3: Earth and a stereographic projection of the earth.

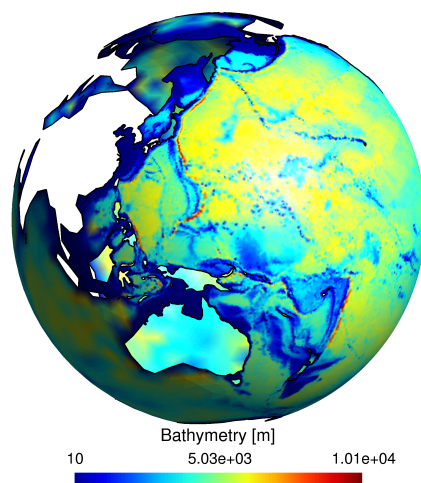


Figure 3.4: Bathymetry of the ocean.

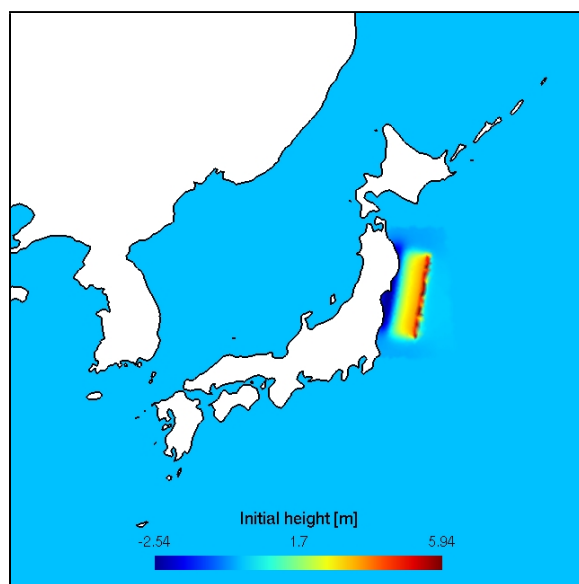


Figure 3.5: Initial condition for η .

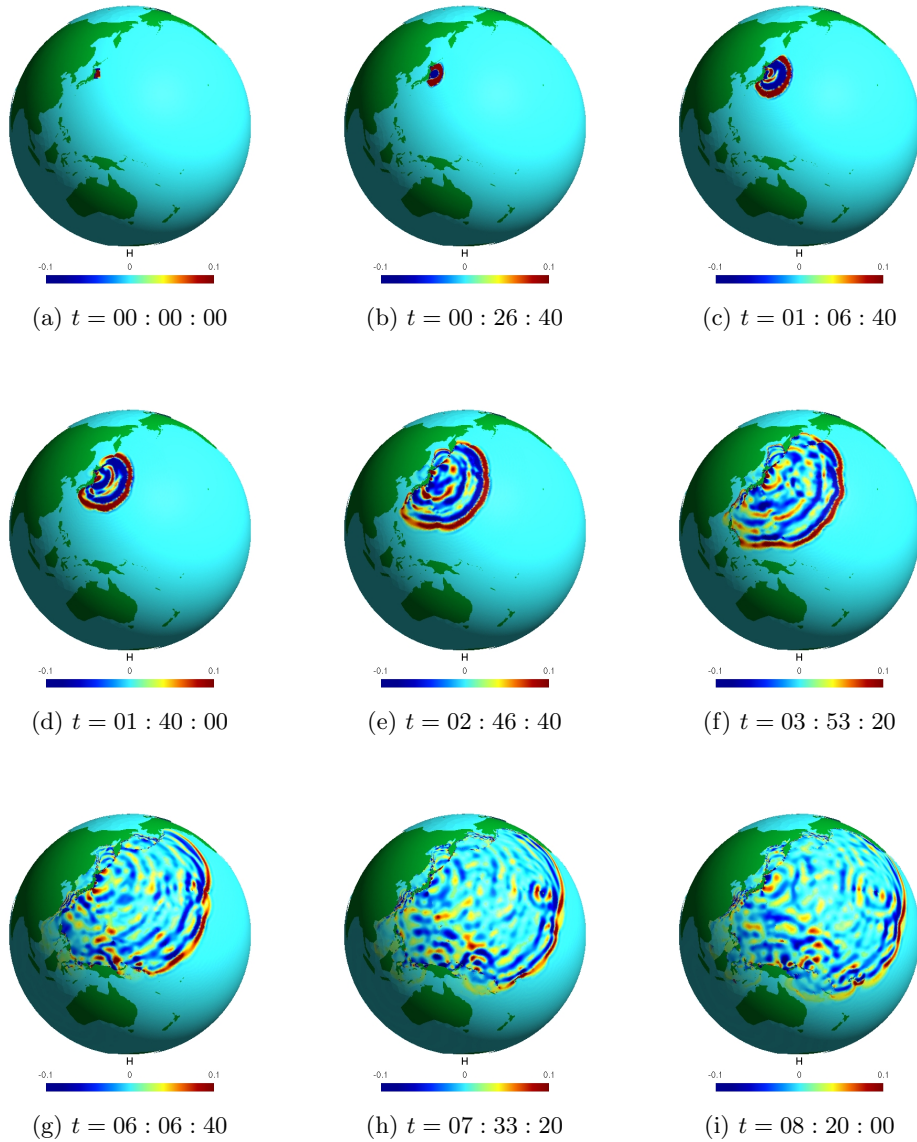


Figure 3.6: Sea surface elevation, η , as a function of time for the tsunami simulation. This simulation is performed our GPU implementation of the DGM method. The color map is cropped at 10 cm for a more representative image.

Chapter 4

Implementing the Discontinuous Galerkin Method on the GPU

In this chapter we examine the general algorithms implemented to solve the equations presented in Chapter 3. After introducing the code's structure, the different kernels and their parallel implementation are defined.

To evaluate GPU performance three different codes were written

- P_{CPU} : This program runs exclusively on the CPU. The code therefore is executed sequentially. The BLAS library is used for the collocation, redistribution and solving steps.
- P_{GPU} : This program runs on the GPU. All kernels are hand written. These kernels have not been fully optimized and are the subject of discussion in Chapter 6.
- P_{GPUBLAS} : This program also runs on the GPU. The collocation and redistribution steps use the CUBLAS library, a BLAS library for CUDA.

The performance of these different implementations are compared in Chapter 5.

4.1 General Algorithms

P_{GPU} and P_{GPUBLAS} can be divided into host driven initialization and device driven resolutions at each time step.

1. Host driven initializations:
 - Read the finite element mesh.
 - Calculate the normal of each interface.

- Evaluate the reference face and reference element nodal functions and their derivatives at the integration points.
- Evaluate the integration weights associated with each face and element integration point.
- Calculate (x, y, z) coordinates in the physical space of the integration points for each element and interface.
- Calculate the Jacobians, Jacobian matrices and their inverses for each element and interface.
- Calculate the inverse mass matrix for each element.
- Initialize the unknowns and intermediate variables on the GPU.
- Send the precalculated data to the GPU

2. Solve the DGM with a Runge-Kutta fourth order integration scheme.

The Runge-Kutta fourth order time integration scheme is implemented in the following way

```

for  $n < N_t$  do {Loop on time steps}
   $U_s = U$  {Store the previous solution}
  for  $k = 0 \rightarrow 3$  do
     $U^* = U_s + \beta_k dU$ 
    DGM to evaluate  $dU = \Delta t \mathbf{M}_e^{-1}(\mathbf{S}_e(U^*, t) + \mathbf{F}_e(U^*, t) + \mathbf{Q}_e(U^*, t))$ 
     $U = U + \gamma_k dU$  {Update the solution}
  end for
end for

```

The DGM is divided into eleven different kernels as defined in Section 4.2. Each kernel defined represents a step in the DGM assembly and resolution process that we presented in Chapter 2.

1. Mapping:

- `mapToFace`: Map element unknowns U to the face unknowns U_F .

2. Collocation:

- `collocationU`: Evaluate U at the element integration points by matrix-matrix multiplication, Equations (2.8) and (2.9).
- `collocationUF`: Evaluate U_F at the face integration points by matrix-matrix multiplication, Equation (2.17).

3. Evaluation:

- `evaluate_sf`: Evaluate the equations defining s and f , Equations (2.11) and (2.12).
- `evaluate_q` : Evaluate the equations defining q , Equation (2.18).

4. Redistribution:

- `jacobian_sf`: Multiply s and f by the Jacobians and inverse Jacobian matrices, Equations (2.13) and (2.14).
- `gemm_sf`: Calculate \mathbf{S} and \mathbf{F} by matrix-matrix multiplication, Equations (2.15) and (2.16).
- `jacobian_q`: Multiply q by the face Jacobians, Equation (2.19).
- `gemm_q`: Calculate \mathbf{Q}_{tcj} by matrix-matrix multiplication, Equation (2.20).

5. Mapping:

- `mapToElem`: Map face contributions, Q_{tcj} , back to the element contributions, \mathbf{Q} .

6. Solving:

- `solve`: Solve the linear system. Multiply each column of \mathbf{S} , \mathbf{F} , \mathbf{Q} by the element mass matrices.

4.2 Kernels and Parallel Implementation

In this section we describe the different kernels defined to solve our problem and their respective parallel implementation. If kernels operate on elements, thread blocks are defined per element: one thread block per element and one thread per node (or integration point) per unknown field. This choice was made in view of optimizing the kernels by placing all the matrices pertaining to the element of the block in the shared memory (Chapter 6). For example, for the `gemm_sf` kernel, there are N_E thread blocks and $N_s N_F$ threads per block. Interfaces are threaded according the same principle: M_T thread blocks, each with $M_s N_F$ threads.

Some code snippets are shown here but the majority of the kernel definitions are in Appendix B.

This is a first implementation and we will refer the reader to Chapter 6 for more information on optimizing these kernels.

Copying a vector to another (equal)

At each Runge-Kutta integration step the solution at the previous time step is stored. Our implementation divides this operation into N_E thread blocks with $N_s \times N_F$ threads per block. Each thread block performs the calculations for the unknown fields at all nodes on the element. P_{GPUBLAS} uses the CUBLAS function, `cublasScopy()`, to perform this operation. A sample kernel code is shown in Section 4.2.

The C function calling the kernel is

```

1 extern "C"
2 void Lgpu_equal(int N_s, int N_E, int N_F, scalar* A, scalar* B)
3 {
4     dim3 dimBlock(N_s, N_F, 1); // Block dimensions
5     dim3 dimGrid(N_E, 1);      // Grid dimensions
6     gpu_equal<<<dimGrid, dimBlock>>>(N_s, N_E, N_F, A, B);
7 }

```

All the calling functions are similar and differ only by the block and grid dimensions. They are omitted in the rest of this chapter.

Adding two vectors (add)

The Runge-Kutta time integration scheme requires the addition of two vectors. Similarly to `equal`, we divided this operation in N_E thread blocks and $N_s \times N_F$ threads per block. P_{GPUBLAS} uses the equivalent CUBLAS function, `cublasSaxpy()`.

Mapping from the element to the face (mapToFace)

Mapping the element unknowns to the interface is done with M_T thread blocks and $M_s \times N_F$ threads per block.

Mapping from the face to the element (mapToElem)

Mapping the face contributions to the element contributions is done with N_E thread blocks and $N_s \times N_F$ threads per block.

Collocation step for the elements (collocationU)

For P_{GPU} , the collocation step for the element unknowns is done with N_E thread blocks and $N_G \times N_F$ threads per block. P_{GPUBLAS} uses the CUBLAS function `cublasSgemm()`. The kernel code is presented in Listing 4.1.

Collocation step for the faces (collocationUF)

The collocation step for the face unknowns is done with M_T thread blocks and $M_G \times N_F$ threads per block. P_{GPUBLAS} uses the CUBLAS function `cublasSgemm()`.

Evaluating s and f (evaluate_sf)

Evaluating the source and flux terms is done with N_E thread blocks and N_G threads per block.

Listing 4.1: collocationU kernel code.

```

1  __global__ void gpu_collocationU(int D, int N_G, int N_s, int
    N_E, int N_F, scalar* Ug, scalar* dUg, scalar* phi, scalar*
    dphi, scalar* U){
2
3  int e = blockIdx.x; // element index
4  int g = threadIdx.x; // integration node index
5  int fc = threadIdx.y; // field index
6
7  scalar sol = 0;
8
9  // Matrix-matrix multiplication for the source contributions
10 for(int i = 0; i < N_s; i++){
11     sol += phi[i*N_G+g] * U[(e*N_F+fc)*N_s+i];
12 }
13 Ug[(e*N_F+fc)*N_G+g] = sol;
14
15 // Matrix-matrix multiplication for the flux contributions
16 sol = 0.0;
17 for(int a = 0; a < D; a++){
18     for(int i = 0; i < N_s; i++){
19         sol += dphi[(i*N_G+g)*D+a] * U[(e*N_F+fc)*N_s+i];
20     }
21     dUg[((e*N_F+fc)*N_G+g)*D+a] = sol;
22     sol = 0.0;
23 }
24 }

```

Evaluating q (evaluate_q)

Evaluating the normal fluxes is done with M_T thread blocks and M_G threads per block.

Multiplying s and f by the Jacobians (jacobian_sf)

Multiplying the source and flux contributions is done with N_E thread blocks and $N_G \times N_F$ threads per block.

Matrix-matrix product to calculate S and F (gemm_sf)

The matrix-matrix multiplication to evaluate the source and flux contributions is done with N_E thread blocks and $N_s \times N_F$ threads per block. P_{GPUBLAS} uses the CUBLAS function `cublasSgemm()`.

Multiplying q by the face Jacobians (jacobian_q)

Multiplying the face terms by the face Jacobians is done with M_T thread blocks and $M_G \times N_F$ threads per block.

Matrix-matrix product to calculate Q_{tcj} (gemm-q)

The matrix-matrix multiplication to evaluate the face contributions is done with M_T thread blocks and $M_s \times N_F$ threads per block. P_{GPUBLAS} uses the CUBLAS function `cublasSgemm()`.

Solving the linear system (solve)

Solving the linear system is done with N_E thread blocks and $N_s \times N_F$ threads per block. It is not possible to call a CUBLAS function from within a kernel. Therefore the matrix vector product in this step does not take advantage of a CUBLAS implementation.

Extension to large problems

As mentioned in Chapter 1, a grid for the Tesla C1060 can contain a maximum of 65535 thread blocks. Because of our threading implementation for most of the kernels, the maximum number of elements allowed in a mesh is $N_E = 65535$. For the tsunami simulations, we changed the kernel parallelism to accommodate more elements per thread block as illustrated in the following code snippet. The variable `blk` is the number of elements per thread block.

```

1 #define blk 3
2
3 // Kernel definitions
4 --global-- void gpu_equal(int N_s, int N_E, int N_F, scalar* A,
5     scalar* B){
6     int e = blockIdx.x*blk+threadIdx.z;
7     if (e < N_E){
8         int i = threadIdx.x;
9         int fc = threadIdx.y;
10
11         A[(e*N_F+fc)*N_s+i] = B[(e*N_F+fc)*N_s+i];
12     }
13 }
14 [...]
15 // Host C functions
16 extern "C"
17 void Lgpu_equal(int N_s, int N_E, int N_F, scalar* A, scalar* B)
18 {
19     int div = N_E/blk;
20     int mod = 0;
21     if (N_E%blk != 0) mod = 1;
22     dim3 dimBlock(N_s, N_F, blk);
23     dim3 dimGrid(div+mod, 1);
24     gpu_equal<<<dimGrid, dimBlock>>>(N_s, N_E, N_F, A, B);
25 }
26 [...]
```


Chapter 5

Comparing CPU and GPU Implementations

The benchmark test for comparing P_{CPU} , P_{GPU} and P_{GPUBLAS} is the simplified shallow water equations, Equation (3.2), with $h_0 = 1$ m and $g_0 = 1$ m/s² on a square domain of size $L = 2$ m. The initial conditions are

$$\begin{aligned}\eta(x, y, t = 0) &= \exp(-2(x^2 - y^2)) \\ u_x(x, y, t = 0) &= 0 \\ u_y(x, y, t = 0) &= 0\end{aligned}$$

The boundaries are considered impermeable. The time evolution of this problem is presented in Appendix A. The problem was integrated over 1000 time steps with $\Delta t = 0.001$ s.

We study the effect of two parameters: the number of elements in the domain, N_E , illustrated in Figure 5.1, and the interpolation order, p , illustrated in Figure 5.2. The kernels are divided so that the number of threads in a thread block increases with p increasing kernel occupancy (Section 5.1.4).

We show increased performance for the GPU implementations as the number of elements and p increase. We use the following color legends in our figures:

- CPU implementation (P_{CPU}) in red and data points with circles;
- GPU hand-coded implementation (P_{GPU}) in green and data points with squares;
- GPU with hand-coded and CUBLAS kernels (P_{GPUBLAS}) in blue and data points with triangles.

In this chapter we present the diagnostics used to compare the different programs (Section 5.1), a preliminary analysis of overall GPU performance (Section 5.2), and a performance analysis of each individual kernel (Section 5.3).

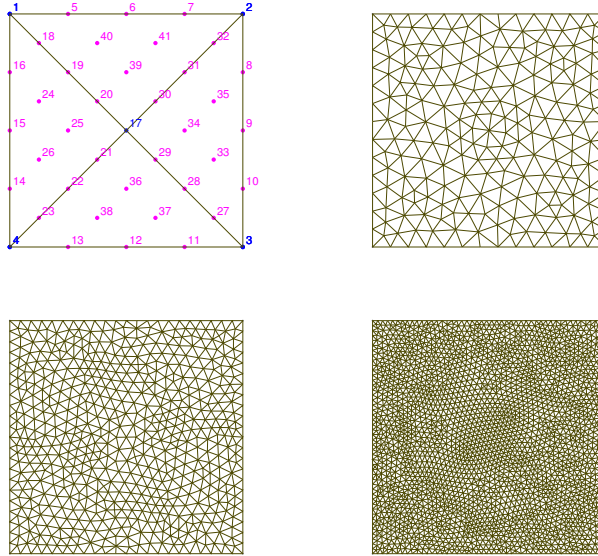


Figure 5.1: Examples of the meshed domain for different numbers of fourth order elements ($p = 4$) and for $N_E = 4$ (top-left), $N_E = 380$ (top-right), $N_E = 1108$ (bottom-left), $N_E = 5072$ (bottom-right). The number of nodes per element, N_s equals 15 for a fourth order element.

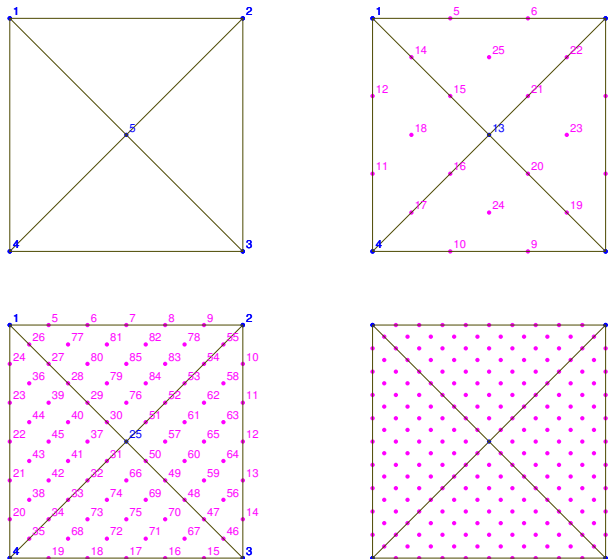


Figure 5.2: Examples of the meshed domain for different element orders with $N_E = 4$ and for $p = 1$ (top-left), $p = 3$ (top-right), $p = 6$ (bottom-left), $p = 10$ (bottom-right). Because of our parallel model, the number of threads per block increases as well, thereby increasing kernel occupancy.

5.1 Diagnostics

Four important metrics are used to evaluate a program's performance: execution time, bandwidth, floating point operations per second and kernel occupancy.

5.1.1 GPU Timers

GPU timers are used to time a kernel's execution. Since kernel launches are asynchronous it is necessary to use `cudaThreadSynchronize()` after the kernel launch to ensure proper timing. `cudaThreadSynchronize()` acts as a barrier and only proceeds with the following host code when the kernel is done executing.

The GPU timer is independent of the computer's operating system. It measures time in milliseconds and has a half microsecond resolution.

5.1.2 Bandwidth

Bandwidth is one of the most important metrics used to evaluate kernel performance and to optimize a kernel. Bandwidth measures the amount of data transferred from global memory to the multiprocessor in a given time interval, measured in GB/s. High bandwidth means that data is being transferred and accessed efficiently.

Kernel bandwidth is particularly dependent on data storage location and how data is accessed. Using the appropriate memory banks and grouping memory fetches are key to increasing bandwidth.

Comparing the device's theoretical bandwidth and the kernel's effective bandwidth is indicative of kernel performance. The device's theoretical bandwidth is the peak theoretical data transferring rate. If the effective bandwidth is much lower than the theoretical bandwidth, the kernel's code should be modified to increase the kernel's effective bandwidth. This is accomplished in several different ways (Chapter 6). In practice, a GPU's real peak bandwidth is lower than the theoretical bandwidth. Running NVIDIA's `bandWidthTest.cu` kernel evaluates the GPU's practical bandwidth. The Tesla C1060's theoretical bandwidth, B_{theo} , is 102.4 GB/s and its practical bandwidth, B_{pract} , is 73.3 GB/s (Appendix D).

Kernel execution time and the amount of data read and written by the kernel are used to calculate the effective bandwidth:

$$B_{\text{eff}} = \frac{B_r + B_w}{T} \quad [GB/s]$$

where B_r is the number of bytes read by the algorithm ($B_r = 4 \text{ B}$ for reading float), B_w is the number of bytes written by the algorithm and T is the kernel execution time in seconds.

We time a kernel execution in the Runge-Kutta integration scheme and average this time, t , by the number of kernel calls. Effective bandwidth for some¹ kernels presented in Section 4.2 are

- equal kernel:

$$\begin{aligned} B_r &= N_s N_E N_F \text{ sizeof(float)} \\ B_w &= N_s N_E N_F \text{ sizeof(float)} \\ B_{\text{eff}} &= \frac{B_r + B_w}{1024^3} \frac{N_t}{t} \end{aligned}$$

- evaluate_sf kernel:

$$\begin{aligned} B_r &= (4N_G N_E + 2) \text{ sizeof(float)} \\ B_w &= (N_G N_E N_F + N_G D N_E N_F) \text{ sizeof(float)} \\ B_{\text{eff}} &= \frac{B_r + B_w}{1024^3} \frac{4N_t}{t} \end{aligned}$$

- evaluate_q kernel:

$$\begin{aligned} B_r &= (2M_T + M_G M_T N_F 2 + 2) \text{ sizeof(float)} \\ B_w &= (M_G M_T N_F 2) \text{ sizeof(float)} \\ B_{\text{eff}} &= \frac{B_r + B_w}{1024^3} \frac{4N_t}{t} \end{aligned}$$

- jacobian_sf kernel:

$$\begin{aligned} B_r &= (N_G N_E N_F + N_G N_E + D D N_G N_E + N_G N_E N_F) \text{ sizeof(float)} \\ B_w &= (N_G N_E N_F + N_G D N_E N_F) \text{ sizeof(float)} \\ B_{\text{eff}} &= \frac{B_r + B_w}{1024^3} \frac{4N_t}{t} \end{aligned}$$

- jacobian_q kernel:

$$\begin{aligned} B_r &= (M_G M_T 2 N_F + M_G M_T) \text{ sizeof(float)} \\ B_w &= (M_G M_T N_F 2) \text{ sizeof(float)} \\ B_{\text{eff}} &= \frac{B_r + B_w}{1024^3} \frac{4N_t}{t} \end{aligned}$$

¹It is hard to define the bandwidth for matrix-matrix multiplications because it is highly dependent on the GPU's internal memory and caches.

5.1.3 Floating Point Operations

The number of floating point operations per second (FLOPS) is an interesting metric for some kernels. The number of floating point operations (flops) for these kernels is

- `collocationU` kernel: This kernel performs two matrix-matrix multiplications.

$$\mathcal{F} = 2N_s N_E N_F N_G + 2N_s N_E N_F N_G D \quad (\text{flops})$$

- `collocationUF` kernel: This kernel performs one matrix-matrix multiplication.

$$\mathcal{F} = 2M_s M_T N_F^2 \quad (\text{flops})$$

- `evaluate_sf` kernel: Two multiplications are done for four components of f .

$$\mathcal{F} = 2N_G D N_E (N_F - 1) \quad (\text{flops})$$

- `jacobian_sf` kernel:

$$\mathcal{F} = N_G N_E N_F + 3D D N_G N_E N_F \quad (\text{flops})$$

- `gemm_sf` kernel:

$$\mathcal{F} = 2N_G N_E N_F N_s + 2N_G D N_E N_F N_s \quad (\text{flops})$$

- `jacobian_q` kernel:

$$\mathcal{F} = M_T M_G N_F^2 \quad (\text{flops})$$

- `gemm_q` kernel:

$$\mathcal{F} = 4M_G M_T N_F \quad (\text{flops})$$

- `solve` kernel:

$$\mathcal{F} = 4N_s N_s N_E N_F \quad (\text{flops})$$

The most expensive steps are the matrix-matrix multiplications which scale as $N_G N_s$ and `solve` kernels.

5.1.4 Kernel occupancy

Kernel occupancy is an apriori diagnostic. Keeping the GPU's multiprocessors as busy as possible is an important aspect of kernel performance. The occupancy metric, a measure of the number of active warps² on a multiprocessor, is a good indicator of multiprocessor resource use. CUDA thread instructions are executed sequentially. Paused or stalled warps, waiting for a memory fetch or a barrier synchronization for example, cause execution latencies. These latencies can be hidden by executing other warps on the multiprocessor: the more active warps on a multiprocessor, the better these latencies are hidden. Occupancy is defined

$$O = \frac{\text{active warps on the multiprocessor}}{\text{potential active warps on the multiprocessor}}$$

As mentioned in Chapter 1, a multiprocessor has a set of registers available for the CUDA threads. This shared resource is allocated among the thread blocks running on the multiprocessor. To maximize the number of thread blocks executing simultaneously on a multiprocessor, thereby increasing occupancy, the CUDA compiler minimizes register usage. Based on register and shared memory requirements, calculating occupancy can help the user pick thread block sizes.

While high occupancy is good, an increase in occupancy does not necessarily mean a corresponding increase in performance. Once a certain occupancy has been reached, the marginal benefit typically associated with an increasing occupancy decreases. Low occupancy is always bad and interferes with CUDA's ability to hide execution latencies.

NVIDIA provides an occupancy calculator in the form of an Excel spreadsheet. The CUDA compiler outputs the number of registers and shared memory used by a given kernel. The calculator uses these numbers to calculate occupancy. In addition, the calculator provides graphs that show the effect of changing the number of threads per block, the number of registers and the amount of shared memory on occupancy (Figure 5.3). The occupancy for the kernels defined in Section 4.2 are presented in Table 5.2. In our particular parallelization, thread block size increases with the element order p . High order runs lead to a higher occupancy (Table 5.2).

Occupancy seems optimal for $p = 7$ and decreases a little for $p = 10$. At $p = 7$, the number of threads per block is such that the multiprocessor's resources are fully used for some kernels.

Most of the kernel's occupancies are block limited. They are limited by the number of blocks of that size allowed on the multiprocessor. They are not limited by the number of registers or by the amount of shared memory available on the multiprocessor. Optimizing the kernels to take advantage of this available memory is done in Chapter 6.

²A warp is a group of 32 threads managed by the multiprocessor.

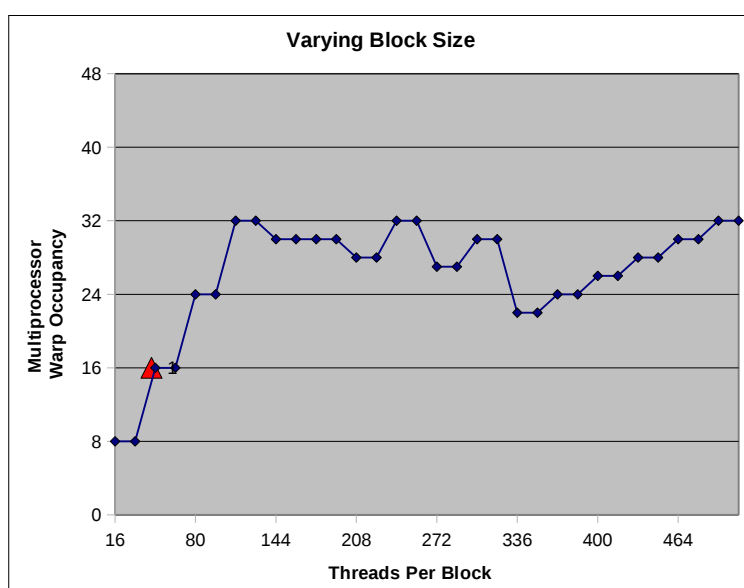


Figure 5.3: Thread block size influencing the occupancy of `gemv.sf`. The red triangle indicates resource usage for a fourth order run. The other data points indicate resource usage for a range of block sizes. Increasing the number of threads per block will increase occupancy.

Kernel	Threads per block	Registers	Shared memory (B)
equal	$N_s N_F$	4	48
mapToFace	$M_s N_F$	8	56
mapToElem	$N_s N_F$	6	56
collocationU	$N_G N_F$	12	80
collocationUF	$M_G N_F$	12	56
evaluate_sf	N_G	10	64
evaluate_q	M_G	15	64
jacobian_sf	$N_G N_F$	12	80
gemm_sf	$N_s N_F$	12	88
jacobian_q	$M_G N_F$	8	56
gemm_q	$M_s N_F$	12	56
solve	$N_s N_F$	14	76

Table 5.1: Thread block sizes, the number of registers and the amount of shared memory used by the different kernels. The compiler returns the number of registers and the amount of shared memory at compilation time. The kernels use relatively few resources. There are 16384 registers and 16384 B of shared memory per multiprocessor. A more appropriate use of these registers and the shared memory would lead to faster kernels (Chapter 6).

order	1	4	7	10
equal	25	50	100	88
mapToFace	25	25	25	50
mapToElem	25	50	100	88
collocationU	25	50	94	88
collocationUF	25	25	25	50
evaluate_sf	25	25	50	94
evaluate_q	25	25	25	25
jacobian_sf	25	50	94	88
gemm_sf	25	50	100	88
jacobian_q	25	25	25	50
gemm_q	25	25	25	50
solve	25	50	100	88

Table 5.2: Occupancy of the kernels for $p = 1, 4, 7, 10$. Occupancy increases with the number of threads per block, which is proportional to p . Most of the important kernels (e.g. collocationU and gemm_sf) show increased occupancy at $p = 7$.

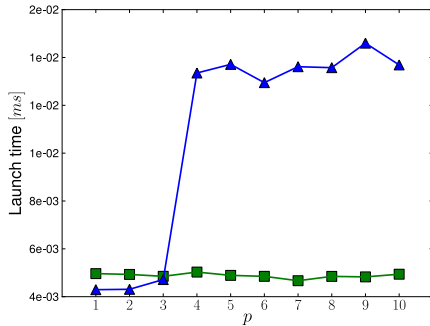
5.2 Preliminary Analysis

GPU timers are used to compare the different DGM implementations. The time taken to transfer the solution from the device to the host and writing the solution to an output file is excluded from the execution times reported here.

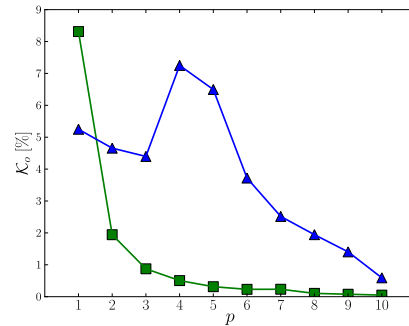
However, overhead from kernel launch time is included. A kernel's launch time is of the order of 10^{-3} ms and it is therefore negligible when compared to the overall kernel execution time (Figure 5.4b). The launch time is independent of the kernel launched and the size of the problem for P_{GPU} . The ratio of kernel overhead time to total kernel execution time decreases with the size of the problem.

$$\mathcal{K}_o = \frac{\text{kernel launch time}}{\text{total kernel execution time}}$$

This is illustrated in Figure 5.4 and in Table C.1. For P_{GPUBLAS} , the kernel launch time varies with the element order. This is most likely because of additional calculations and memory setup that the CUBLAS library does to optimize the matrix-matrix product.



(a) Launch time for `collocationU` as a function of element order for $N_E = 1108$.



(b) Ratio of launch time to total execution time for `collocationU` as a function of element order for $N_E = 1108$.

Figure 5.4: Launch times are small compared to overall kernel execution time. The ratio of launch time to total execution time decreases with the problem size. For P_{GPU} , the launch time does not depend on the problem size. For P_{GPUBLAS} , this overhead varies with the element order, most likely because CUBLAS optimizes block sizes and memory accesses for the matrix-matrix product. This additional negligible overhead associated to setting up the problem enables high speedup for the CUBLAS library.

The times reported here are averaged over one thousand time steps. Most of the kernels were launched four times per integration step because

we used a Runge-Kutta fourth order algorithm. Each kernel was timed and averaged over four thousand executions to reduce execution time variance. These variances are small and average times reported can be trusted. We measure the relative standard deviation, defined as

$$RSD = \frac{\text{standard deviation of the execution time}}{\text{average execution time}} \quad (\%)$$

and illustrated for P_{GPU} in Figure 5.5 and in Table C.2. The relative standard deviation decreases with p and is small for the kernels of interest, e.g. the matrix-matrix products (Figure 5.5).

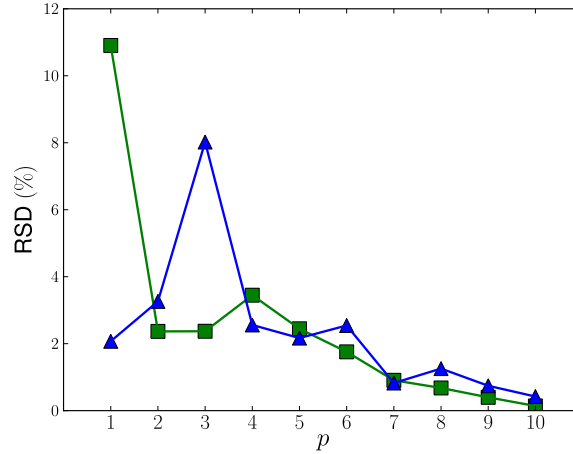


Figure 5.5: Low execution time relative standard deviation is ensured by averaging over one thousand Runge-Kutta integration steps. We show here the relative standard deviation for `collocationU` as a function of the element order for $N_E = 1108$. For P_{GPUBLAS} and P_{GPU} , the relative standard deviation decreases with increasing element order.

A first comparison between P_{CPU} , P_{GPU} and P_{GPUBLAS} illustrates the performance gains of the GPU implementations. We compare the average execution time of one Runge-Kutta iteration for P_{CPU} , P_{GPU} and P_{GPUBLAS} in Figures (5.6) and (5.7). The time for one CPU integration step increases linearly with the number of elements. The same observation can be made for P_{GPU} and P_{GPUBLAS} once the number of elements is larger than 100. GPU overhead is then negligible compared to overall computation time. To measure P_{GPU} and P_{GPUBLAS} speedup we calculate the ratio

$$S = \frac{\text{CPU time}}{\text{GPU time}}$$

and present it in Figures (5.6b) and (5.7b).

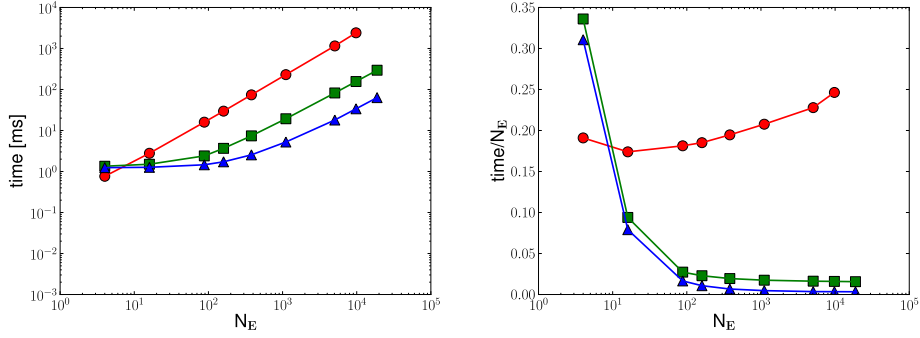
For $p = 4$ and increasing numbers of elements, P_{GPU} is asymptotically fifteen times faster than BLAS P_{CPU} (Figure 5.6). P_{GPU} and P_{GPUBLAS} reach their asymptotic speedup quickly, around $N_E = 100$ (Figure 5.6a). In the asymptotic regime the speedup does not increase significantly with the number of elements anymore. We use $N_E = 1108$ to compare polynomial orders to ensure that the asymptotic regime is reached. P_{GPUBLAS} is asymptotically seventy times faster than P_{CPU} . The large speedup difference between P_{GPU} and P_{GPUBLAS} is due to the CUBLAS library's optimal use of the GPU's architecture to compute the matrix-matrix products.

For $N_E = 1108^3$ and $p = 5$ or 6 , P_{GPU} is ten times faster than P_{CPU} whereas P_{GPUBLAS} is almost fifty times faster (Figure 5.7). The CUBLAS library is well tuned for matrix-matrix products. P_{GPU} 's speedup decreases with the element order because the hand-coded matrix-matrix products are not optimized. The reader can refer to Chapter 6 for code improvements that might help increase GPU speedup. Increasing kernel occupancy by increasing p (and therefore increasing the number of threads per block) does not seem to help for all the kernels. Some kernels show increasing speedup as p increases while others do not; this was discussed during our analysis of the individual kernels (Section 5.3). P_{GPUBLAS} 's speedup peaks around $p = 5$, an element order frequently used in simulations. We show in Section 5.3 that the `solve` kernel's speedup decreases with element order. The `solve` operation dominates P_{GPUBLAS} 's execution time at high orders, thereby decreasing P_{GPUBLAS} 's speedup at high p .

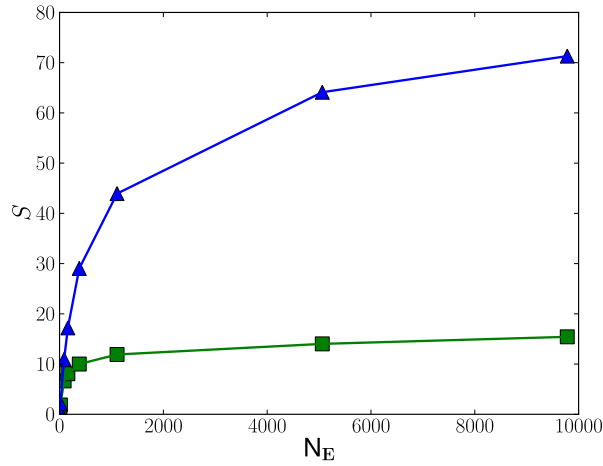
We have shown high speedup with our GPU implementations. Our hand-coded version is ten to fifteen times faster than the CPU program which uses the BLAS library. Using the CUBLAS library increases this speedup to about fifty.

Some GPU methods will show a net improvement over their CPU counterparts because their algorithms are more appropriate for the GPU. It is important to see how these methods scale with the problem size parameters, N_E and p , to understand the differences between P_{CPU} , P_{GPU} and P_{GPUBLAS} .

³We are in the asymptotic regime where increasing the number of elements does not increase speedup significantly.

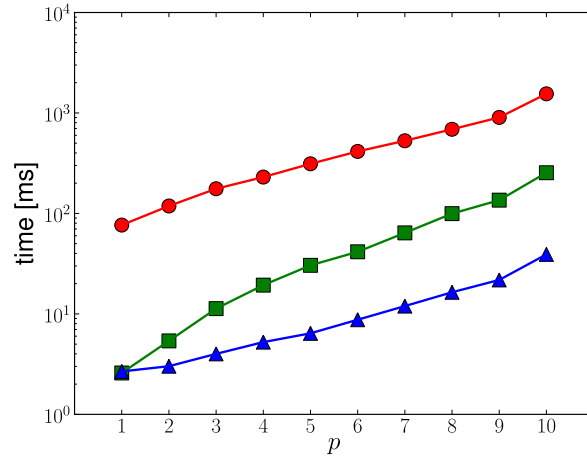


(a) Average time for one Runge-Kutta iteration. P_{CPU} time increases linearly with N_E . P_{GPU} and $P_{GPUBLAS}$ integration time is dominated by overhead below $N_E = 100$. For higher N_E , P_{GPU} and $P_{GPUBLAS}$ time increases linearly with N_E , though with different slopes. An asymptotic regime is reached for N_E larger than 100.

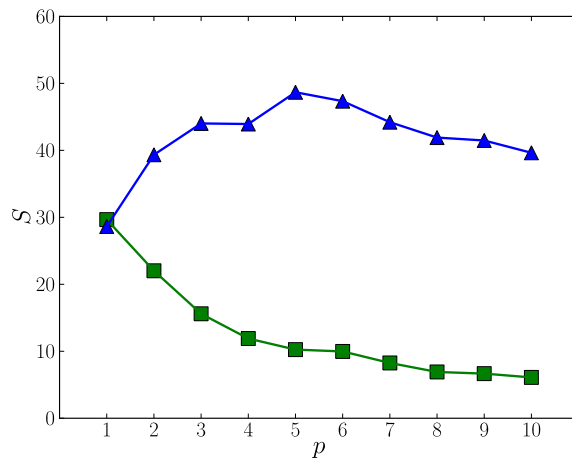


(b) Speedup, $S = \frac{\text{CPU time}}{\text{GPU time}}$, for P_{GPU} and $P_{GPUBLAS}$. P_{GPU} is asymptotically fifteen times faster than P_{CPU} . $P_{GPUBLAS}$ is asymptotically seventy times faster than P_{CPU} .

Figure 5.6: Average time and speedup for one Runge-Kutta iteration for P_{CPU} , P_{GPU} and $P_{GPUBLAS}$ as a function of N_E . The element order is fixed ($p = 4$).



(a) Average time for one Runge-Kutta iteration. P_{CPU} time increases rapidly with the element order. This increase is much larger than the one for P_{GPU} and $P_{GPUBLAS}$.



(b) Speedup as a function of p . For $p = 5$ and $p = 6$, P_{GPU} is about ten times faster than P_{CPU} whereas $P_{GPUBLAS}$ is forty-eight times faster. Decrease in speedup at high order is due to inefficient matrix-matrix products for P_{GPU} and inefficient solving for $P_{GPUBLAS}$.

Figure 5.7: Average time and speedup for one Runge-Kutta iteration for P_{CPU} , P_{GPU} and $P_{GPUBLAS}$ as a function of element order, p . The number of elements in the mesh is fixed ($N_E = 1108$).

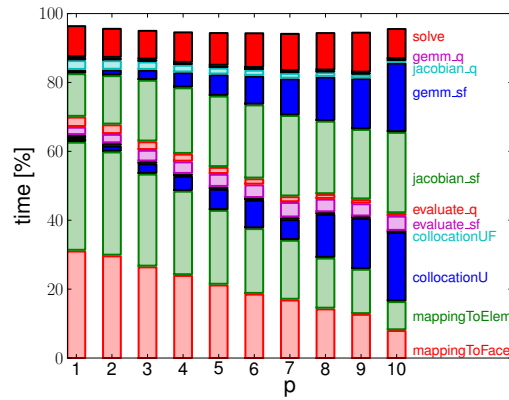
5.3 Individual Kernel Analysis

It is important to analyze the different kernels to understand which kernels are efficient and well implemented on the GPU (Figure 5.8). We compare the fraction of time spent in the kernels for one Runge-Kutta integration. These percentages do not add up to 100% because the `add` and `equal` operations are not illustrated. Faded rectangles in the bar graphs are operations that do not use a BLAS or CUBLAS libraries. Dark color rectangles in the bar graphs are operations performed with the BLAS or CUBLAS libraries. Analysis for fixed numbers of elements and increasing p is illustrated in Figure 5.8. The corresponding figure for fixed p and an increasing number of elements is Figure C.1.

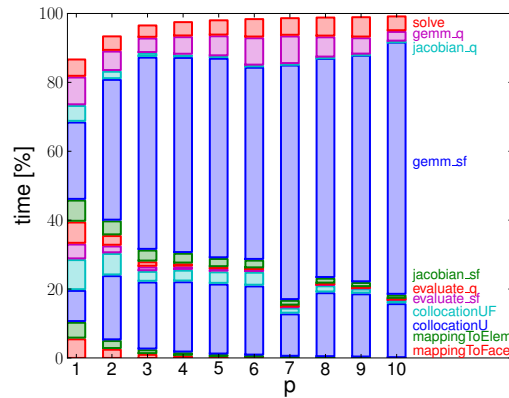
For P_{CPU} in Figure 5.8a, it is clear that the expensive operations are the matrix-matrix multiplications and Jacobian multiplications. The matrix-matrix operations for `collocation` and `gemm` are done efficiently with the BLAS library. For increasing p , the matrix-matrix products, while still efficient, dominate the total execution time. We stated in Chapter 2 that evaluating the physics, while not efficient, is not as costly as the other operations. Evaluating the physics for P_{CPU} is only a small fraction of the total time.

For P_{GPU} , the expensive operations are the matrix-matrix products performed in `collocationU` and `gemm_sf` (Figure 5.8b). These products are hand-coded and do not take full advantage of the GPU's architecture. Using different memory types as well as changing the code's structure could presumably solve this problem (Chapter 6). The percentage of time spent in `collocationUF`, `jacobian_sf`, `jacobian_q`, `gemm_q` and `solve` does not increase significantly with p . They seem to scale well with an increasing number of elements. The matrix-matrix products for the interface contributions, `collocationUF` and `gemm_q`, are smaller than the products for the element contributions because the number of nodes on an interface is much lower than the number of nodes on an element. Multiplying by the Jacobians is very efficient. As expected, the fraction of time spent evaluating the physics of the problem is negligible. For $p = 7$, kernel occupancy of the matrix-matrix multiplications is maximal (Table 5.2). The difference between `collocationU` for $p = 7$ and the other orders is most likely due to the high occupancy at $p = 7$ (100% occupancy). This kernel's occupancy for $p = 10$ is only 88%.

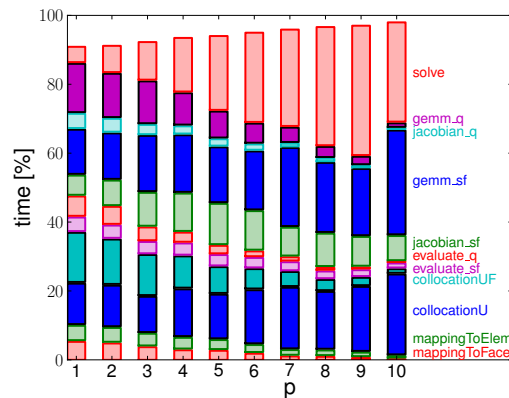
For P_{GPUBLAS} , using the CUBLAS library ensures efficient matrix-matrix multiplications (Figure 5.8c). As opposed to P_{GPU} , these operations scale well with the number of elements and do not dominate calculation time. There seems to be an optimal size for the matrix-matrix product in `gemm_sf` before $p = 10$ (Figure C.11b). At high order the `solve` operations dominate the total time. The matrix-vector product is not implemented with the CUBLAS



(a) For P_{CPU} the BLAS implementations ensure efficient matrix-matrix products.



(b) For P_{GPU} , the matrix-matrix products dominate the total execution time.



(c) For P_{GPUBLAS} , the matrix-matrix products are efficiently calculated with CUBLAS. The `solve` operations dominate the total execution time for high p .

Figure 5.8: Comparing the fraction of time spent in each kernel for the different implementations as a function of element order. The number of elements is fixed: $N_E = 1108$. Dark color rectangles are operations that use the BLAS or CUBLAS libraries.

library⁴.

Multiplying by the Jacobians involves many small nested loops whose size, while not known at computation time, can only be 1, 2, or 3, the spatial dimension. In other codes it has been shown that unrolling these specific loops significantly increases performances for the Jacobian multiplications (e.g. `jacobian_sf`). However, loop unrolling was not performed in this implementation.

Investigating the individual kernels and comparing the different implementations help understand where GPU implementations perform well. We will discuss the `equal` kernel, the mappings, the physics evaluations, the matrix-matrix products, and the linear system solve.

5.3.1 Equalizing Two Vectors

The `equal` operation, which copies the content of one vector into another, is programmed in three different ways:

- P_{CPU} 's `equal` is a sequential loop iterating on each element of the vectors one at a time;
- P_{GPU} is threaded so that each thread copies one element from one vector to the other;
- P_{GPUBLAS} uses the `cublasScopy` function.

A complete list of figures for this kernel can be found in Figure C.2.

As previously mentioned, execution time is dominated by kernel overhead at a low number of elements. For a high number of elements and high p the average execution time slopes are very different, indicating that the average time for P_{CPU} 's `equal` will increase at a faster rate than for P_{GPU} and P_{GPUBLAS} (Figure C.2a).

This kernel achieves high speedup (Figure 5.9). P_{GPU} is asymptotically twenty-five times faster than P_{CPU} . P_{GPUBLAS} is thirty-five times faster. Even for a simple copy operation the CUBLAS library is faster than our GPU implementation.

The effective bandwidth for this operation with P_{GPUBLAS} approaches the practical bandwidth (Figure 5.9b). This is not the case for our GPU implementation. The difference between these two implementations can be easily explained by CUBLAS's optimized kernels.

5.3.2 Mappings

Mapping from the element to the interface and vice-versa is very lightweight operation. The fraction of time spent doing one of these mappings is around

⁴A CUBLAS function cannot be called from within a kernel. The only way to use the CUBLAS matrix-vector product is to call it sequentially from the host code: multiplying with CUBLAS each different inverse mass matrix with the element's contributions one at a time. This obviously would not take advantage of the GPU's architecture.

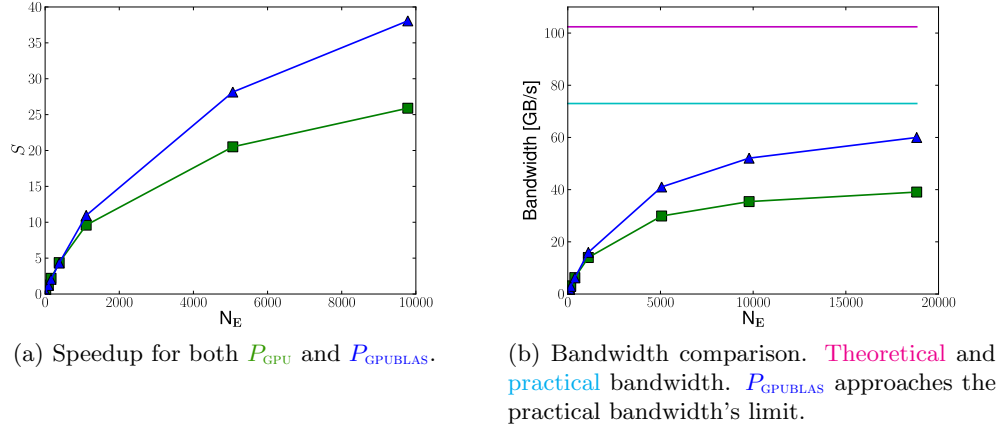


Figure 5.9: `equal` kernel performance as a function of N_E with fixed element order ($p = 4$) for P_{GPU} , $P_{GPUBLAS}$.

0.5% for high order elements. These operations are well threaded. The number of nodes that are mapped to and from the faces is much smaller than the number of nodes in the mesh because the nodes lying inside the elements are not mapped. The mappings achieve very high speedup (Figure 5.10). There is a large increase in speedup for $p = 7$. We have mentioned previously that for $p = 7$ occupancy seems optimal and this most likely explains the increase in speedup. Optimal memory alignment could also explain this increase.

While the same kernel was used for P_{GPU} and $P_{GPUBLAS}$, $P_{GPUBLAS}$ speedup is a little higher (Figure 5.10). We are not able to explain this slight difference as of the writing of this report.

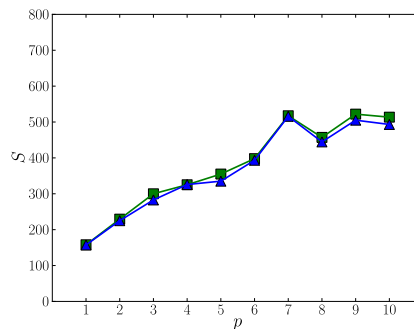
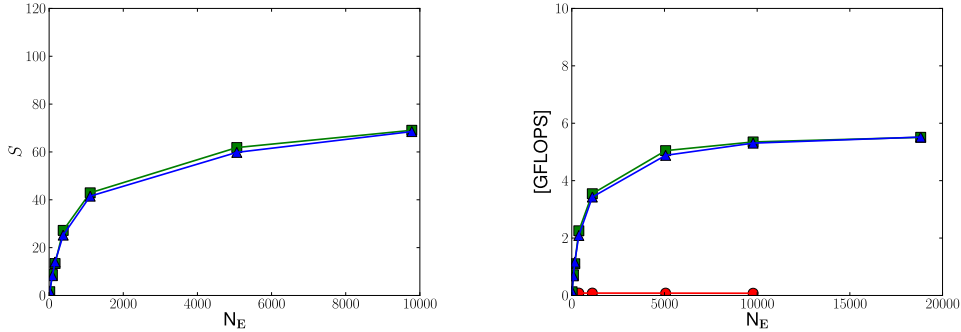


Figure 5.10: `mapToFace` kernel performance as a function of element order. This kernel, identical for P_{GPU} and $P_{GPUBLAS}$ achieves high speedup.

5.3.3 Evaluating the Physics

Evaluating the physics is not a costly operation. For $p = 4$, only 1% of the total time in a Runge-Kutta iteration step is spent evaluating the source and flux contributions.

The GPU implementation⁵ achieves high speedup. It is almost sixty times faster than P_{CPU} (Figure 5.11). The number of floating point operations per second for the GPU implementation is much greater than that for the CPU implementation (Figure 5.11b). However, this number is far from the theoretical limit of the Tesla C1060 (almost 1 TFLOPS). This is most likely because the kernel is bandwidth limited. Memory access is slowing down the kernel's performance.



(a) High speedup for both P_{GPU} and P_{GPUBLAS} . (b) Floating point operations per second is much greater for the GPU implementation. However, this number is far from the theoretical limit of the Tesla C1060 (almost one TFLOPS).

Figure 5.11: `evaluate_sf` kernel performance as a function of N_E with fixed element order ($p = 4$) for P_{CPU} , P_{GPU} , P_{GPUBLAS} .

5.3.4 Matrix-matrix Products

The greatest speedup factor comes from using the CUBLAS library. The matrix-matrix products are the costliest operations and dominate execution time. The largest matrix-matrix products are calculated in the `collocationU` and `gemm_sf` steps.

The number of floating point operations per second for P_{GPU} and P_{GPUBLAS} increases with the element order (Figure 5.12). However, this number increases very differently for the two implementations. For P_{GPU} , this number is close to 20 GFLOPS for high p and for P_{GPUBLAS} close to 80 GFLOPS. P_{GPU} does not optimize block sizes and memory access. All the data for

⁵Identical for P_{GPU} and P_{GPUBLAS}

P_{GPU} lies in the global memory which suffers from high latency. For $p = 7$, we notice a jump in floating point operations per second for P_{GPU} , suggesting that we have reached an optimal occupancy or that the kernel is doing coalesced memory fetches (Chapter 6). P_{GPUBLAS} achieves high floating point operations per second because it optimizes block sizes and memory access. However, it is still far from the theoretical peak performance of the Tesla C1060.

For these large matrix-matrix products P_{GPUBLAS} , designed to use the GPU's resources efficiently, is about thirty times faster than P_{CPU} (Figure 5.13). The BLAS library used by P_{CPU} accomplishes these operations efficiently and rivals our hand-coded GPU implementation. We suggest improvements to P_{GPU} in Chapter 6 to increase its performance.

The matrix-matrix products for the face terms of a 2D problem do not achieve very high speedup. While P_{GPUBLAS} is eight to ten times faster than P_{CPU} , these products are not large enough to show remarkable performance increases on the GPU. However, generalizing this method to 3D problems will ensure that the face contribution matrices are much larger. We therefore expect significant speedup for 3D problems. In addition, we are solving an explicit formulation of the DGM. An implicit formulation would further increase the mass matrices and the number of operations in the solve step.

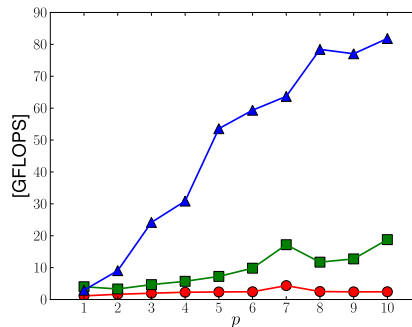


Figure 5.12: `collocationU` kernel floating point operations per second as a function of p with fixed number of elements ($N_E = 1108$) for P_{CPU} , P_{GPU} , P_{GPUBLAS} . P_{GPU} suffers from high memory latency. P_{GPUBLAS} offers a very high number of floating point operations per second.

5.3.5 Solving the Linear System

Solving the linear system is done by multiplying the element's inverse mass matrix with the element's contributions. We consider the general case of a mesh composed of high order curved elements where each element has a different inverse mass matrix. Consequently, this operation cannot be expressed as a single matrix-matrix product. P_{CPU} loops over each element

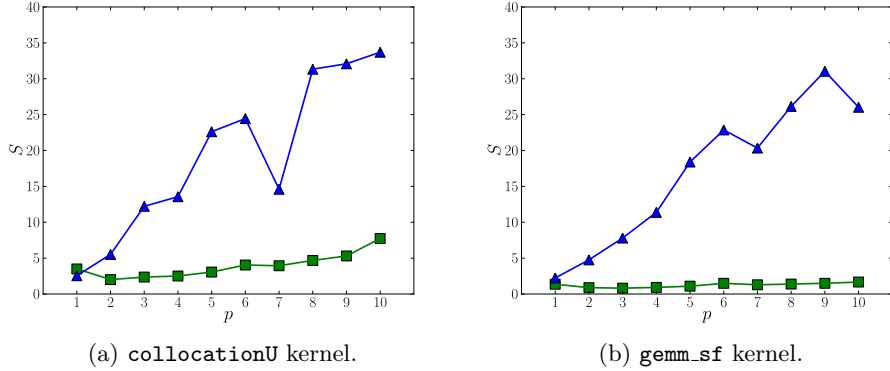


Figure 5.13: Speedup for P_{GPU} and $P_{GPUBLAS}$ as a function of element order ($N_E = 1108$). Using the CUBLAS library makes $P_{GPUBLAS}$ about thirty times faster than P_{CPU} . P_{GPU} is limited by memory access and is not much faster than P_{CPU} , which uses the BLAS library.

individually and uses the BLAS library to perform this matrix-vector operation. The GPU implementation does not use the CUBLAS library because a CUBLAS function cannot be called from within a kernel.

Our GPU implementation is asymptotically thirty times faster than P_{CPU} for a high number of elements (Figure 5.14a). Because of our implementation, increasing the number of elements increases the number of blocks but does not increase the work load of each block.

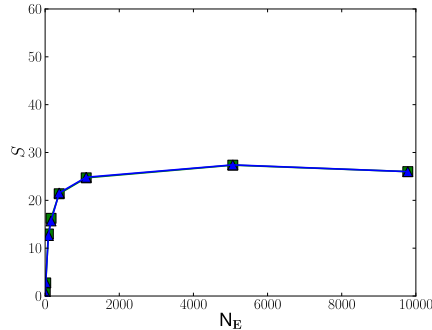
Speedup decreases as a function of the element order (Figure 5.14b). The number of threads per block increases such that this kernel slows down with the element order. Each block is accessing the element's inverse mass matrix multiple times, once for each unknown field instead of storing it in the shared memory. For lack of time we were not able to implement this improvement. The size of the matrix-vector product also increases with element order.

Solving the linear system dominates execution time for $P_{GPUBLAS}$ (Figure 5.8c). This step accounts for twenty percent of the $P_{GPUBLAS}$'s execution time at high element order, decreasing $P_{GPUBLAS}$'s speedup (Figure 5.7b). Improving this kernel is a high priority improvement which we discuss in Chapter 6.

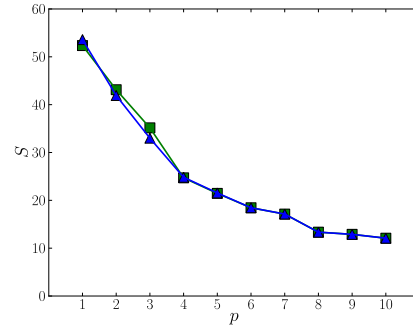
5.4 Conclusion

In this chapter we have shown high speedup with the GPU implementation: approximately fifty times faster for $P_{GPUBLAS}$ and around ten to fifteen times faster for P_{GPU} at $p = 5$.

Speedup tends to saturate at a high number of elements. The number



(a) Speedup as a function of number of elements ($p = 4$).



(b) Speedup as a function of element order ($N_E = 1108$).

Figure 5.14: Speedup for P_{GPU} and P_{GPUBLAS} . Decreasing speedup for high order elements is due to inefficiently implemented matrix-vector products. The appropriate use of shared memory should increase the speedup for high p .

of elements defines the number of blocks and the grid size. The amount of operations done per block does not vary with the number of elements.

Most kernels show increased speedup as p increases. Increasing the element order increases the number of threads per block and the amount of work done per block. As mentioned in Chapter 4, increased occupancy generally implies an increase in performance. Another important factor that could explain this increased performance is that global memory fetches are better coalesced for high orders. At $p = 7$, hand-coded kernel occupancy is optimal for many kernels and we have shown an increase in speedup for this order.

Evaluating the physics, multiplying by the Jacobians, performing the mappings and copying two vectors are operations that are well suited to the GPU.

P_{GPU} is limited by slow memory access and therefore does not achieve great speedup for the matrix-matrix products. The BLAS library rivals our implementation for the smaller matrix-matrix products. P_{GPUBLAS} achieves very high speedup for the matrix-matrix product because it is designed to optimize this type of operation.

Solving the linear system is the least efficient kernel because its performance decreases as the element order increases. The matrix-vector product is inefficiently implemented. This could be averted with a more appropriate use of the GPU's shared memory.

We did not achieve the theoretical bandwidth and peak performance limits of the GPU, leading us to believe that these kernels can be optimized. Improving the GPU kernels is the subject of the next chapter.

Chapter 6

GPU Code Improvements

Despite notable performance improvements, P_{GPU} presents only a first, relatively naive implementation of the DGM. In this chapter we present some general guidelines to optimizing code and suggest several high priority improvements to our current implementation.

These improvements were not implemented. Though we assume they will speed up the code we do not have results to prove this. These improvements should be quantified using the appropriate timers, the kernel bandwidth, floating point operations per second and occupancy.

The CUBLAS library has a remarkable performance for the matrix-matrix multiplication operations (`gemm_sf`, `collocationU`). Optimizing a hand-made code for these operations might therefore be a waste of time.

6.1 High Priority Improvement Guidelines

It is essential to use the memory as close to the multiprocessors as possible. The objective is to minimize global memory fetches. The use of shared memory is one of the most important performance enhancers. As mentioned in Chapter 1 this memory resides on the multiprocessor and it is accessed about a hundred times faster than global memory. Minimizing global memory use and maximizing shared memory use is therefore extremely important.

It is important to keep in mind that shared memory is divided into banks. Multiple threads simultaneously accessing to the same bank can lead to bank conflicts. Conflicting accesses are serialized and decrease kernel performance.

Global memory fetches are done by half-warps of threads (16 threads on the Tesla C1060). A coalesced fetch means that all threads in a half-warp access the global memory of the device in one transaction. Coalescing global memory fetches can increase kernel performance by several orders of magnitude. A coalesced fetch can be ensured if certain guidelines are followed (NVIDIA Corporation, 2009).

Using cached constant memory is another performance enhancer. Constant memory on the Tesla C1060 is relatively small, about 65 kB.

After optimizing memory data storage locations and data fetches execution configuration should be optimized. The objective is to keep the multiprocessors as busy as possible by increasing occupancy. Executing other warps on the multiprocessor to hide latencies caused by a paused or stalled warp increases code performance.

We should note that it is essential to minimize data transfers from the host to the device. These transfers are slow and costly but in our case it is not really a problem since all the calculations are done on the device. Host to device transfer is 4.7 GB/s on the Tesla C1060 compared to a device to device transfer of 73.3 GB/s (theoretically 102.4 GB/s). Most often it is a good idea to keep calculations on the device to avoid transferring data back to the host even if these calculations do not show remarkable improvement. P_{GPU} therefore copies all necessary data from the host to the device once. The only data transfers from the device to the host occur when the device sends the solution back to the host after a defined number of integration steps. These transfers are excluded from our timing procedures.

6.2 Suggested Improvements to P_{gpu}

Depending on availability, constant memory should be considered for storing the constant matrices such as the Jacobians, the inverse mass matrices, the nodal functions, and the mappings.

In the kernels multiplying by the Jacobians, `jacobian_sf` and `jacobian_q`, the Jacobians should be placed in the shared memory of the block. Each thread block corresponds to one element and therefore the Jacobians and inverse Jacobians are the same for each block.

Loop unrolling should be done whenever possible. Iterations on the space dimensions or sides of an interface should be explicit and hard-coded.

Optimizing the `solve` kernel should be a priority. It greatly reduces P_{GPUBLAS} 's performance and its speedup decreases as a function of the element order. Placing the element's inverse mass matrix and the associated column of the matrices that store the physical law evaluated at the integration points, \mathbf{S} , \mathbf{F} , and \mathbf{Q} , in the block's shared memory will increase the matrix-vector product performance used to solve the linear system. A more efficient algorithm for the matrix-vector product could be implemented.

Block multiplying two matrices and using the shared memory for this product greatly increases performance. However, the CUBLAS library might still be faster than block multiplying matrices by hand.

We have shown that most of our kernels have very low occupancy for low p (Table 5.2). Grouping multiple elements on a same block would increase occupancy for lower polynomial orders. This does not necessarily

mean increased kernel performance. Additional speedup could be achieved by finding the appropriate number of elements per block, either empirically or with the occupancy calculator. For example, for $p = 4$, the `gemm_sf` kernel's occupancy can be increased from 50% to 75% by placing two elements per block instead of one. Grouping three elements on one block increases the occupancy to 94%. This optimization might not be necessary for a 3D problem. We expect a much higher occupancy for a 3D problem since interfaces and volumes will contain many more nodes, increasing the number of threads per block.

Measuring the effects of these improvements on code performance would be very interesting. This list of improvements is non-exhaustive. Other improvements could of course be implemented though global memory coalescing¹ and shared memory use should be given high priority.

6.3 Alternative Discontinuous Galerkin Implementation

We propose a different kind of kernel structure which may increase code performance. The objective is to increase register and shared memory use because access to these memories is extremely fast (Section 1.4). Instead of dividing the DGM into small, separate kernels, these kernels would be grouped. Each thread block would use shared memory and registers as much as possible and eliminate transfers from global memory.

A `scatter` operation in the kernels would allow all thread blocks to fetch the necessary data from the global memory and store them locally: the element's relevant Jacobians, inverse mass matrices, and nodal functions.

Five kernels would be necessary:

1. `mapToFace` to map from the element to the interface unknowns. This is the same kernel as defined in Section 4.2.
2. `calculate_sf` to calculate the source and flux contributions \mathbf{S} and \mathbf{F} . This kernel would be threaded in N_E thread blocks, each containing N_F threads.
3. `calculate_q` to calculate the face contributions Q_{tcj} . This kernel would be threaded in M_T thread blocks, each containing N_F threads.
4. `mapToElem` to map from the interface to the element contributions. This is the same kernel as defined in Section 4.2.
5. `solve` to solve the linear system, also defined previously in Section 4.2.

¹The correct memory alignments could ensure coalesced fetches, discussed in depth for the DGM in (Warburton *et al.*, 2009).

`calculate_sf`'s algorithm would be

```

1 -- global-- void gpu.SFcalculation(args){
2     int e = blockIdx.x;
3     int fc= threadIdx.x;
4     // Fetch the data relevant to the element and store them in
5     // share memory (jacobians, inverse mass matrix, nodal
6     // functions,...)
7     // Collocation (matrix-vector product)
8     // Evaluation
9     // Redistribution (matrix-vector product)
10    // Send S_e and F_e to the global memory
11 }

```

Despite the many advantages of this program structure (use of fast shared memory and registers) there are a few disadvantages.

The number of registers and the amount of shared memory on a multiprocessor is limited. The suggested kernels would use many more registers and more shared memory than the small kernels currently implemented. Limited by resource availability, this could reduce the number of thread blocks concurrently executing on a multiprocessor and severely reduce occupancy. Thread blocks using too many registers could be forced to push data onto local memory which is as slow as global memory.

Another disadvantage is that these kernels are less parallel. A block of threads representing one element would only contain N_F threads. Each thread would do the calculations for all the nodes in each unknown field of the element. Sequential loops in each thread would be necessary to treat all the nodes of the element and all the integration points of the element. The matrix-vector product in the collocation and redistribution steps would be done sequentially.

Avoiding this last disadvantage is possible. Each thread in a block could do the calculations for each node of the element or each integration point. There would be therefore $\max(N_s, N_G) \times N_F$ threads per block for `calculate_sf`. Shared memory use and index calculation is more complicated.

This implementation illustrates the compromise between maximizing fast memory usage and minimizing register use to increase occupancy and maximizing the code's parallel structure.

Conclusions

In this dissertation we implemented a parallel numerical method, the Discontinuous Galerkin method, to solve partial differential equations on a graphics processing unit.

GPUs have a unique architecture composed of many multiprocessors which can handle thousands of lightweight threads concurrently, and many different types of memory. Understanding the architecture of the GPU is essential to an intelligent implementation of the numerical method that takes full advantage of the GPU's resources.

The DGM is a numerical method that considers the elements individually. It is therefore intrinsically parallel, making it ideal for the GPU. The DGM is divided into three steps: collocation, evaluation and redistribution. It is shown in Lambrechts (2011) that the first and third steps can be rewritten as expensive matrix-matrix products that can be efficiently calculated with a BLAS routine. Evaluating the physics of the problem, the second DGM step, does not require as many operations as the collocation and redistribution steps but it cannot be efficiently calculated.

We have shown considerable speedup for the DGM method with the GPU. The GPU hand-coded implementation is overall ten to fifteen times faster than a CPU version using the BLAS library. With the use of CUBLAS, the GPU's optimized BLAS library, the GPU implementation is about fifty times faster than the CPU version. The hand-coded kernels do not use the GPU's shared memory. The matrix-matrix products therefore dominate execution time at high orders whereas they are efficiently calculated with the CUBLAS library. Increasing occupancy by increasing the number of threads per block seems to have a positive effect on GPU performance. A poor implementation of the matrix-vector product in the linear system solve greatly slows down the GPU implementation which uses CUBLAS.

Taking better advantage of the GPU's architecture will increase the GPU's performance. Using fast access shared memory and registers and block multiplying matrices will significantly speed up the kernels. Implementing global memory coalesced fetches and optimizing occupancy by placing several elements per thread block should be explored.

Today's scientists are using parallel computer clusters and parallel algorithms to solve increasingly complex problems. High-performance scientific

computation has been looking to use the modern GPU, a parallel cluster on its own and traditionally tailored to gaming and visualization, for scientific purposes. In this paper we have shown how an unoptimized implementation of the DGM can achieve considerable speedup.

Bibliography

- Bernard, Paul-Emile. 2008. *Discontinuous Galerkin methods for geophysical flow modeling*. Ph.D. thesis, Université catholique de Louvain.
- Dawson, Clint, & Proft, Jennifer. 2004. Coupled discontinuous and continuous Galerkin finite element methods for the depth-integrated shallow water equations. *Computer Methods in Applied Mechanics and Engineering*, **193**(3-5), 289 – 318.
- Kubatko, Ethan J., Westerink, Joannes J., & Dawson, Clint. 2006. hp Discontinuous Galerkin methods for advection dominated problems in shallow water flow. *Computer Methods in Applied Mechanics and Engineering*, **196**(1-3), 437 – 451.
- Lambrechts, Jonathan. 2011. *Finite Element Methods for Coastal Flows: Application to the Great Barrier Reef*. Ph.D. thesis, Université catholique de Louvain.
- NVIDIA Corporation. 2008. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation.
- NVIDIA Corporation. 2009. *NVIDIA CUDA C Programming Best Practices Guide*. NVIDIA Corporation.
- NVIDIA Corporation. 2010. *NVIDIA CUDA C Programming Guide*. NVIDIA Corporation.
- Remacle, Jean-François, Frazão, Sandra Soares, Li, Xiangrong, & Shephard, Mark S. 2006. An adaptive discretization of shallow-water equations based on discontinuous Galerkin methods. *International Journal for Numerical Methods in Fluids*, **52**(8), 903–923.
- Slaoui, Karim. 2011. Working paper: derivations of the tsunami equations in stereographic coordinates.
- Vos, Peter E.J., Sherwin, Spencer J., & Kirby, Robert M. 2010. From h to p efficiently: Implementing finite and spectral/hp element methods to achieve optimal performance for low- and high-order discretisations. *Journal of Computational Physics*, **229**(13), 5161 – 5181.

- Warburton, Tim, Bridge, Jeffrey, & Hesthaven, Jan S. 2009. Nodal discontinuous Galerkin methods on graphics processors. *Journal of Computational Physics*, **228**, 7863–7882.

Appendix A

Solution to the Benchmark Problem

The solution to the benchmark test as a function of time is illustrated in Figure (A.1). The initial conditions are

$$\begin{aligned}\eta(x, y, t = 0) &= \exp(-2x^2 - 2y^2) \\ u_x(x, y, t = 0) &= 0 \\ u_y(x, y, t = 0) &= 0\end{aligned}$$

The boundaries are impermeable and $h_0 = 1$ m and $g_0 = 1$ m/s².

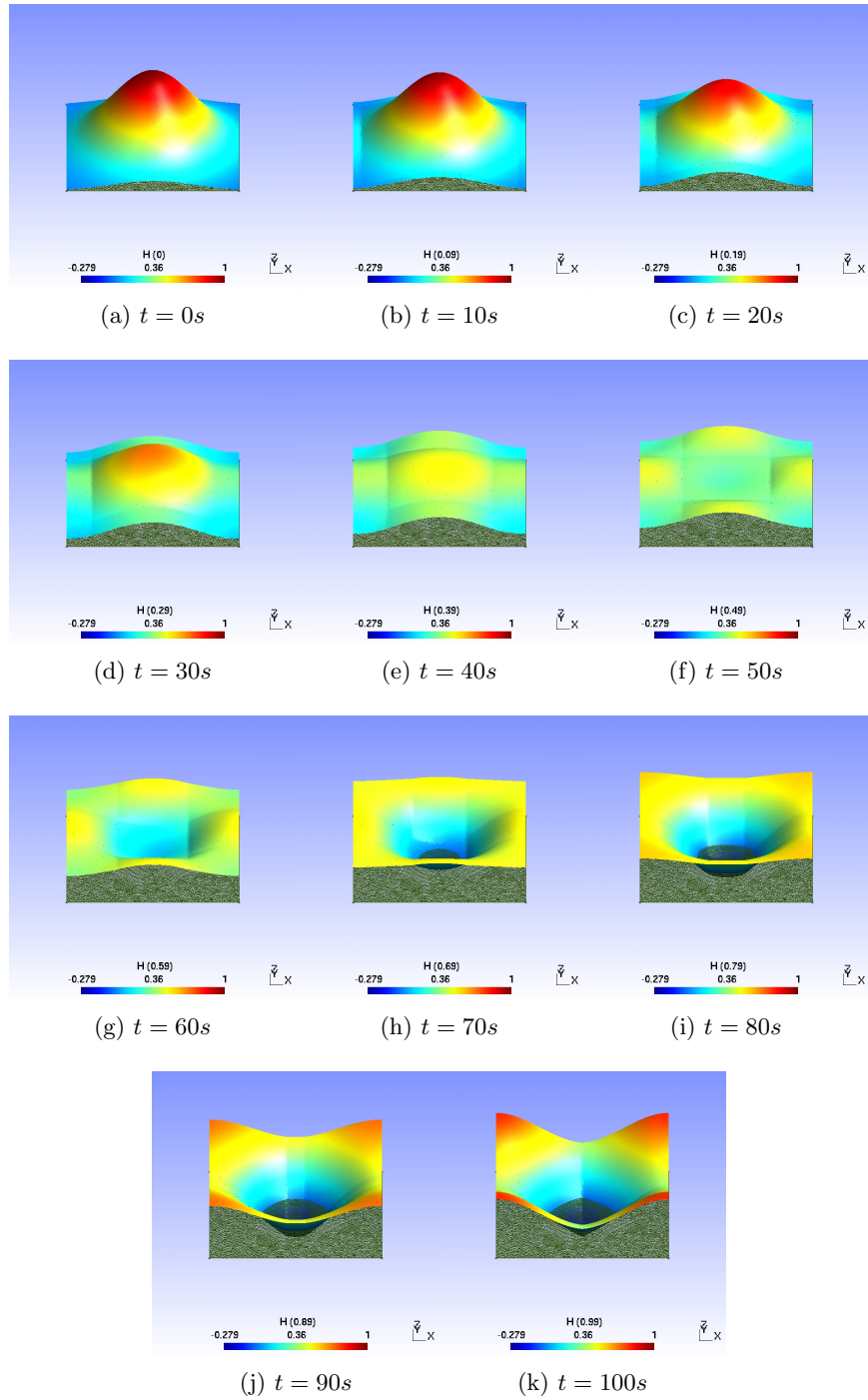


Figure A.1: Relative water elevation, η as a function of time with $h_0 = 1$ m and $g_0 = 1$ m/s².

Appendix B

Kernel Codes

Copying a vector to another (equal)

At each Runge-Kutta integration step the solution at the previous time step is stored. Our implementation divides this operation into N_E thread blocks with $N_s \times N_F$ threads per block. Each thread block performs the calculations for the unknown fields at all nodes on the element. P_{GPUBLAS} uses the CUBLAS function, `cublasScopy()`, to perform this operation.

```
1 __global__ void gpu_equal(int N_s, int N_E, int N_F, scalar* A,
2   scalar* B){
3   int e = blockIdx.x;
4   int i = threadIdx.x;
5   int fc= threadIdx.y;
6
7   A[(e*N_F+fc)*N_s+i] = B[(e*N_F+fc)*N_s+i];
8 }
```

The C function calling the kernel is

```
1 extern "C"
2 void Lgpu_equal(int N_s, int N_E, int N_F, scalar* A, scalar* B)
3   {
4   dim3 dimBlock(N_s, N_F, 1);
5   dim3 dimGrid(N_E, 1);
6   gpu_equal<<<dimGrid, dimBlock>>>(N_s, N_E, N_F, A, B);
7 }
```

All the calling functions are similar and differ only by the block and grid dimensions. We omit them in the rest of this chapter.

Adding two vectors (add)

The Runge-Kutta time integration scheme requires the addition of two vectors. Similarly to `equal`, we divided this operation in N_E thread blocks and

$N_s \times N_F$ threads per block. P_{GPUBLAS} uses the equivalent CUBLAS function, `cublasSaxpy()`.

```

1  __global__ void gpu_add(int N_s, int N_E, int N_F, scalar* A,
2     scalar* B, scalar c){
3     // A = A + c*B
4     int e = blockIdx.x;
5     int i = threadIdx.x;
6     int fc = threadIdx.y;
7
8     A[(e*N_F+fc)*N_s+i] = A[(e*N_F+fc)*N_s+i] + c*B[(e*N_F+fc)*
9     N_s+i];
}

```

Mapping from the element to the face (mapToFace)

Mapping the element unknowns to the interface is done with M_T thread blocks and $M_s \times N_F$ threads per block.

```

1  __global__ void gpu_mapToFace(int M_s, int M_T, int N_F, int*
2     map, scalar* U, scalar* UF){
3     int t = blockIdx.x;
4     int j = threadIdx.x;
5     int fc = threadIdx.y;
6
7     int idx = -1;
8     int face;
9
10    for(int d = 0; d < 2; d++){
11        face = ((t*N_F+fc)*2+d)*M_s+j;
12        idx = map[face];
13        if(idx != -1){
14            UF[face] = U[idx];
15        }
16        else if (idx == -1){
17            if (fc == 0) UF[((t*N_F+fc)*2+1)*M_s+j] = UF[((t*N_F+
18                fc)*2+0)*M_s+j]; // eta
19            else if (fc == 1) UF[((t*N_F+fc)*2+1)*M_s+j] = -UF[((t*N_F+
20                fc)*2+0)*M_s+j]; // ux
21            else if (fc == 2) UF[((t*N_F+fc)*2+1)*M_s+j] = -UF[((t*N_F+
22                fc)*2+0)*M_s+j]; // uy
23        }
24    }
25 }

```

Mapping from the face to the element (mapToElem)

Mapping the face contributions to the element contributions is done with N_E thread blocks and $N_s \times N_F$ threads per block.

```

1  __global__ void gpu_mapToElement(int N_s, int N_E, int N_F, int*
    invmap, scalar* Q, scalar* Qtcj){
2
3  int e = blockIdx.x;
4  int i = threadIdx.x;
5  int fc= threadIdx.y;
6  int idx = 0;
7
8  scalar sol = 0;
9
10 for(int k = 0; k < 2; k++){
11     idx = invmap[((e*N_F+fc)*2+k)*N_s+i];
12     if(idx != -1){
13         sol += Qtcj[idx];
14     }
15 }
16 Q[(e*N_F+fc)*N_s+i] = sol;
17 }

```

Collocation step for the elements (collocationU)

The collocation step for the element unknowns is done with N_E thread blocks and $N_G \times N_F$ threads per block. P_{GPUBLAS} uses the CUBLAS function `cublasSgemm()`.

```

1  __global__ void gpu_collocationU(int D, int N_G, int N_s, int
    N_E, int N_F, scalar* Ug, scalar* dUg, scalar* phi, scalar*
    dphi, scalar* U){
2
3  int e = blockIdx.x;
4  int g = threadIdx.x;
5  int fc= threadIdx.y;
6
7  scalar sol = 0;
8
9  for(int i = 0; i < N_s; i++){
10     sol += phi[i*N_G+g] * U[(e*N_F+fc)*N_s+i];
11 }
12 Ug[(e*N_F+fc)*N_G+g] = sol;
13
14 sol = 0.0;
15 for(int a = 0; a < D; a++){
16     for(int i = 0; i < N_s; i++){
17         sol += dphi[(i*N_G+g)*D+a] * U[(e*N_F+fc)*N_s+i];
18     }
19     dUg[((e*N_F+fc)*N_G+g)*D+a] = sol;
20     sol = 0.0;
21 }
22 }

```

Collocation step for the faces (collocationUF)

The collocation step for the face unknowns is done with M_T thread blocks and $M_G \times N_F$ threads per block. P_{GPUBLAS} uses the CUBLAS function `cublasSgemm()`.

```

1 --global-- void gpu_collocationUF(int M_G, int M_s, int M_T, int
    N_F, scalar* UgF, scalar* psi, scalar* UF){
2
3     int t = blockIdx.x;
4     int g = threadIdx.x;
5     int fc= threadIdx.y;
6
7     scalar sol = 0;
8     for(int d = 0; d < 2; d++){
9         for(int j = 0; j < M_s; j++){
10            sol += psi[j*M_G+g] * UF[((t*N_F+fc)*2+d)*M_s+j];
11        }
12        UgF[((t*N_F+fc)*2+d)*M_G+g] = sol;
13        sol = 0.0;
14    }
15 }

```

Evaluating s and f (evaluate_sf)

Evaluating the source and flux terms is done with N_E thread blocks and N_G threads per block.

```

1 --global-- void gpu_evaluate_sf(int D, int N_G, int N_E, int N_F
    , scalar* s, scalar* f, scalar* Ug, scalar H0, scalar G0){
2
3     int e = blockIdx.x;
4     int g = threadIdx.x;
5
6     scalar eta = Ug[(e*N_F+0)*N_G+g];
7
8     s[(e*N_F+0)*N_G+g] = 0;
9     s[(e*N_F+1)*N_G+g] = 0;
10    s[(e*N_F+2)*N_G+g] = 0;
11
12    // Flux derive par rapport a x
13    f[((e*N_F+0)*N_G+g)*D+0] = H0*Ug[(e*N_F+1)*N_G+g]; // u_x
14    f[((e*N_F+1)*N_G+g)*D+0] = G0*eta; // eta
15    f[((e*N_F+2)*N_G+g)*D+0] = 0;
16
17    // Flux derive par rapport a y
18    f[((e*N_F+0)*N_G+g)*D+1] = H0*Ug[(e*N_F+2)*N_G+g]; // u_y
19    f[((e*N_F+1)*N_G+g)*D+1] = 0;
20    f[((e*N_F+2)*N_G+g)*D+1] = G0*eta; // eta
21 }

```

Evaluating q (evaluate_q)

Evaluating the normal fluxes is done with M_T thread blocks and M_G threads per block.

```

1  __global__ void gpu_evaluate_q(int M_G, int M_T, int N_F, scalar
    * q, scalar* UgF, scalar H0, scalar G0, scalar* normals){
2
3  int t = blockIdx.x;
4  int g = threadIdx.x;
5
6  scalar nx = normals[t*2+0];
7  scalar ny = normals[t*2+1];
8  scalar etaL= UgF[((t*N_F+0)*2+0)*M_G+g];
9  scalar etaR= UgF[((t*N_F+0)*2+1)*M_G+g];
10 scalar uLn = UgF[((t*N_F+1)*2+0)*M_G+g] * nx + UgF[((t*N_F+2)
    *2+0)*M_G+g] * ny;
11 scalar uRn = UgF[((t*N_F+1)*2+1)*M_G+g] * nx + UgF[((t*N_F+2)
    *2+1)*M_G+g] * ny;
12
13 scalar h0 = H0;
14 scalar g0 = G0;
15
16 // first equation
17 scalar qL = -0.5*h0*(uLn + uRn + sqrt(g0/h0)*(etaL-etaR)); //
    Left
18 q[((t*N_F+0)*2+0)*M_G+g] = qL;
19 q[((t*N_F+0)*2+1)*M_G+g] = -qL;
20 // second
21 qL = -0.5*g0*nx*(etaL+etaR+sqrt(h0/g0)*(uLn-uRn)); // Left
22 q[((t*N_F+1)*2+0)*M_G+g] = qL;
23 q[((t*N_F+1)*2+1)*M_G+g] = -qL;
24 // third
25 qL = -0.5*g0*ny*(etaL+etaR+sqrt(h0/g0)*(uLn-uRn)); // Left
26 q[((t*N_F+2)*2+0)*M_G+g] = qL;
27 q[((t*N_F+2)*2+1)*M_G+g] = -qL;
28 }

```

Multiplying s and f by the Jacobians (jacobian_sf)

Multiplying the source and flux contributions is done with N_E thread blocks and $N_G \times N_F$ threads per block.

```

1  __global__ void gpu_jacobian_sf(int D, int N_G, int N_E, int N_F
    , scalar* sJ, scalar* fJ, scalar* s, scalar* f, scalar* J,
    scalar* invJac){
2
3  int e = blockIdx.x;
4  int g = threadIdx.x;
5  int fc= threadIdx.y;
6
7  scalar j = J[e*N_G+g];
8  scalar sol = 0.0;

```

```

9
10 sJ [(e*N_F+fc)*N_G+g] = s [(e*N_F+fc)*N_G+g] * j;
11 for(int alpha = 0; alpha < D; alpha++){
12     for(int a = 0; a < D; a++){
13         sol += invJac [(e*N_G+g)*D+alpha]*D+a] * f [(e*N_F+fc)*N_G+g
14             )*D+a] * j;
15     }
16     fJ [(e*N_F+fc)*N_G+g]*D+alpha] = sol;
17     sol = 0;
18 }

```

Matrix-matrix product to calculate S and F (gemm_sf)

The matrix-matrix multiplication to evaluate the source and flux contributions is done with N_E thread blocks and $N_s \times N_F$ threads per block. P_{GPUBLAS} uses the CUBLAS function `cublasSgemm()`.

```

1 --global-- void gpu_gemm_sf(int D, int N_G, int N_s, int N_E,
2     int N_F, scalar* S, scalar* F, scalar* sJ, scalar* fJ, scalar
3     * phi_w, scalar* dphi_w){
4
5     int e = blockIdx.x;
6     int i = threadIdx.x;
7     int fc = threadIdx.y;
8
9     scalar sol = 0.0;
10
11     // S = phi_w.transpose() x sJ
12     for(int g = 0; g < N_G; g++){
13         sol += phi_w[i*N_G+g] * sJ [(e*N_F+fc)*N_G+g];
14     }
15     S[(e*N_F+fc)*N_s+i] = sol;
16     sol = 0.0;
17
18     // F = dphi_w.transpose() x fJ
19     sol = 0.0;
20     for(int g = 0; g < N_G; g++){
21         for(int a = 0; a < D; a++){
22             sol += dphi_w[(i*N_G+g)*D+a] * fJ [(e*N_F+fc)*N_G+g]*D+a];
23         }
24     }
25     F[(e*N_F+fc)*N_s+i] = sol;
26     sol = 0.0;
27 }

```

Multiplying q by the face Jacobians (jacobian_q)

Multiplying the face terms by the face Jacobians is done with M_T thread blocks and $M_G \times N_F$ threads per block.

```

1  __global__ void gpu_jacobian_q(int MG, int MT, int NF, scalar
    * qJ, scalar * q, scalar * JF){
2
3  int t = blockIdx.x;
4  int g = threadIdx.x;
5  int fc= threadIdx.y;
6
7  for(int d = 0; d < 2; d++){
8      qJ[((t*N_F+fc)*2+d)*MG+g] = q[((t*N_F+fc)*2+d)*MG+g] * JF
        [((t*MG+g)*2+d)];
9  }
10 }

```

Matrix-matrix product to calculate Q_{tcj} (gemm-q)

The matrix-matrix multiplication to evaluate the face contributions is done with M_T thread blocks and $M_s \times N_F$ threads per block. P_{GPUBLAS} uses the CUBLAS function `cublasSgemm()`.

```

1  __global__ void gpu_gemm_q(int MG, int M_s, int M_T, int N_F,
    scalar * Qtcj, scalar * qJ, scalar * psi_w){
2
3  int t = blockIdx.x;
4  int j = threadIdx.x;
5  int fc= threadIdx.y;
6
7  scalar sol = 0.0;
8
9  // Qtcj = psi_w.transpose() x qJ
10 for(int d = 0; d < 2; d++){
11     for(int g = 0; g < MG; g++){
12         sol += psi_w[j*MG+g] * qJ[((t*N_F+fc)*2+d)*MG+g];
13     }
14     Qtcj[((t*N_F+fc)*2+d)*M_s+j] = sol;
15     sol = 0.0;
16 }
17 }

```

Solving the linear system (solve)

Solving the linear system is done with N_E thread blocks and $N_s \times N_F$ threads per block. It is not possible to call a CUBLAS function from within a kernel. Therefore the matrix vector product in this step does not take advantage of a CUBLAS implementation.

```

1  __global__ void gpu_solve(int N_s, int N_E, int N_F, scalar * DU,
    scalar * S, scalar * F, scalar * Q, scalar * Minv, scalar Dt){
2
3  int e = blockIdx.x;
4  int i = threadIdx.x;

```

```
5  int fc= threadIdx.y;
6
7  scalar sol = 0.0;
8
9  for(int ii = 0; ii < N_s; ii++){
10     sol += Minv[(e*N_s+ii)*N_s+i]*(S[(e*N_F+fc)*N_s+ii] + F[(e*
        N_F+fc)*N_s+ii] + Q[(e*N_F+fc)*N_s+ii]);
11 }
12 DU[(e*N_F+fc)*N_s+i] = Dt*sol;
13 sol = 0.0;
14 }
```


Appendix C

Additional Figures and Tables

C.1 Launch Time Tables

Kernel	$N_E = 4$		$N_E = 1108$		$N_E = 18822$	
	ms	\mathcal{K}_o [%]	ms	\mathcal{K}_o [%]	ms	\mathcal{K}_o [%]
equal	0.0045	25.19	0.0042	15.75	0.0053	3.27
mapToFace	0.0037	18.14	0.0039	8.99	0.0046	1.12
mapToElem	0.0039	20.18	0.0040	8.48	0.0044	0.90
collocationU	0.0049	11.79	0.0049	0.50	0.0051	0.03
collocationUF	0.0039	16.90	0.0046	2.81	0.0046	0.20
evaluate_sf	0.0046	25.46	0.0047	9.93	0.0052	1.01
evaluate_q	0.0043	21.30	0.0044	10.43	0.0047	1.18
jacobian_sf	0.0054	24.36	0.0049	3.33	0.0050	0.23
gemm_sf	0.0057	9.85	0.0052	0.19	0.0057	0.01
jacobian_q	0.0038	20.55	0.0042	12.25	0.0045	1.56
gemm_q	0.0040	17.39	0.0039	1.53	0.0043	0.11
solve	0.0047	17.30	0.0047	2.23	0.0050	0.14

Kernel	$p = 1$		$p = 5$		$p = 10$	
	ms	\mathcal{K}_o [%]	ms	\mathcal{K}_o [%]	ms	\mathcal{K}_o [%]
equal	0.0044	17.57	0.0041	13.61	0.0045	8.32
mapToFace	0.0035	9.16	0.0038	7.97	0.0042	6.49
mapToElem	0.0044	13.90	0.0043	8.48	0.0044	4.82
collocationU	0.0047	7.64	0.0052	0.33	0.0051	0.05
collocationUF	0.0043	7.30	0.0048	1.78	0.0048	0.74
evaluate_sf	0.0049	15.94	0.0051	9.09	0.0051	3.11
evaluate_q	0.0048	11.18	0.0045	10.43	0.0046	10.39
jacobian_sf	0.0047	10.90	0.0048	2.44	0.0051	0.69
gemm_sf	0.0055	3.70	0.0054	0.12	0.0068	0.01
jacobian_q	0.0042	13.29	0.0043	11.23	0.0048	8.50
gemm_q	0.0040	7.41	0.0044	0.97	0.0042	0.22
solve	0.0045	12.79	0.0047	1.33	0.0049	0.17

Table C.1: Launch times and ratio of kernel overhead time to total kernel execution time, \mathcal{K}_o , for fixed p and increasing N_E and for fixed N_E and increasing p for P_{GPU} .

C.2 Relative Standard Deviation Tables

Kernel	$RSD(\%)$ for $p = 4$		
	$N_E = 4$	$N_E = 1108$	$N_E = 18822$
equal	11.41	24.17	4.46
mappingToFace	9.67	17.48	1.44
mappingToElem	24.81	15.60	1.32
collocationU	4.92	3.45	2.13
collocationUF	9.86	3.97	0.28
evaluate_sf	9.11	12.79	1.29
evaluate_q	13.99	17.74	1.53
jacobian_sf	11.21	4.38	0.31
gemm_sf	20.30	0.41	1.21
jacobian_q	42.43	17.26	2.27
gemm_q	26.28	2.30	0.16
solve	11.92	3.20	0.33
rk_step	1.94	0.36	0.43

Kernel	$RSD(\%)$ for $N_E = 1108$		
	$p = 1$	$p = 5$	$p = 10$
equal	28.92	24.17	9.87
mapToFace	15.94	17.48	10.61
mapToElem	21.91	15.60	4.12
collocationU	10.91	3.45	0.14
collocationUF	10.46	3.97	1.79
evaluate_sf	24.03	12.79	4.55
evaluate_q	15.76	17.74	14.80
jacobian_sf	17.77	4.38	1.23
gemm_sf	3.77	0.41	0.18
jacobian_q	21.53	17.26	12.07
gemm_q	16.71	2.30	0.29
solve	26.61	3.20	0.45
rk_step	1.11	0.36	0.07

Table C.2: Execution time relative standard deviation, $RSD = \frac{\sigma}{\mu}$ for fixed $p = 4$ and increasing N_E and for fixed $N_E = 1108$ and increasing p for P_{GPU} . The relative standard deviations are small. Therefore the average execution times reported in this paper are accurate.

C.3 Comparing Kernels

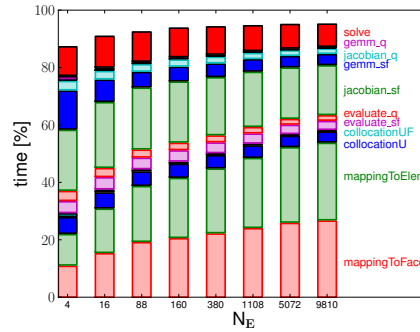
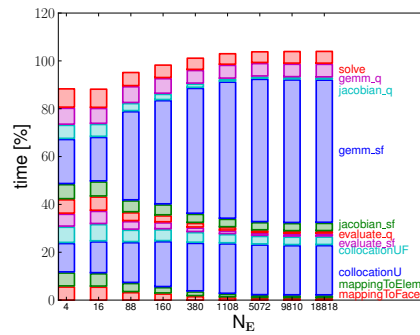
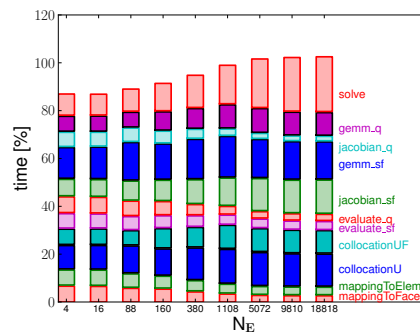
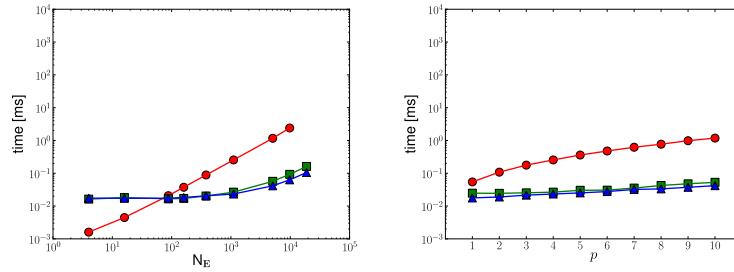
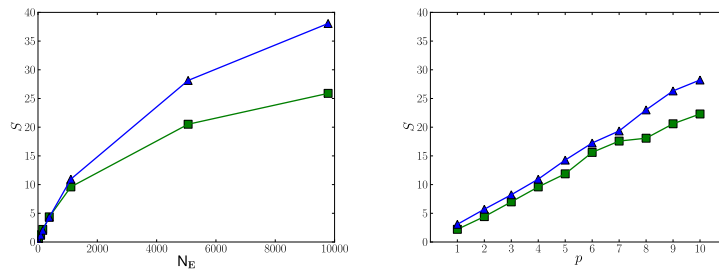
(a) P_{CPU} (b) P_{GPU} (c) P_{PUBLAS}

Figure C.1: Comparing the percentage of time spent in each kernel for the different implementations as a function of the number of elements. The element order is fixed: $p = 4$. Solid color rectangles are operations that use the BLAS or CUBLAS libraries.

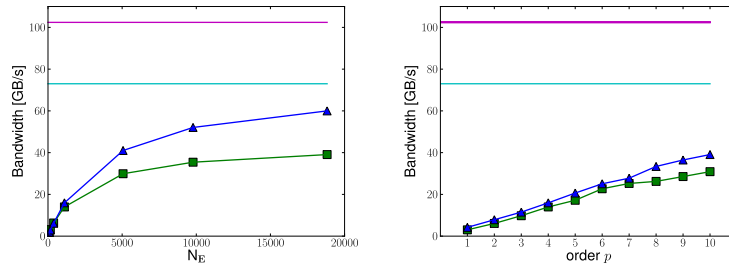
C.3.1 equal



(a) Average kernel execution time.



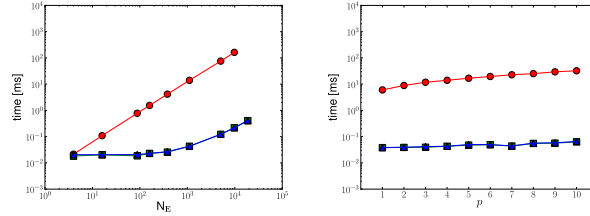
(b) Speedup.



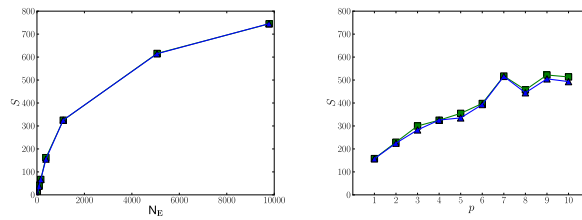
(c) Bandwidth comparison. Theoretical and practical bandwidth.

Figure C.2: equal kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , P_{GPUBLAS} .

C.3.2 Mappings

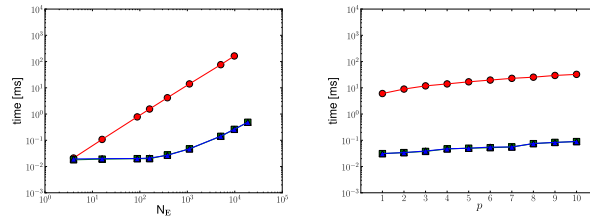


(a) Average kernel execution time.

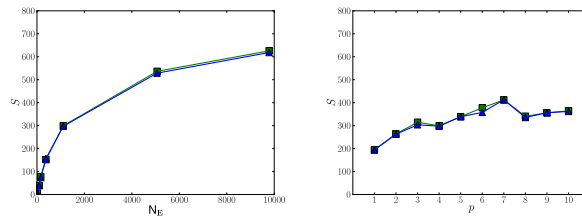


(b) Speedup.

Figure C.3: `mappingToFace` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , $P_{GPUBLAS}$.



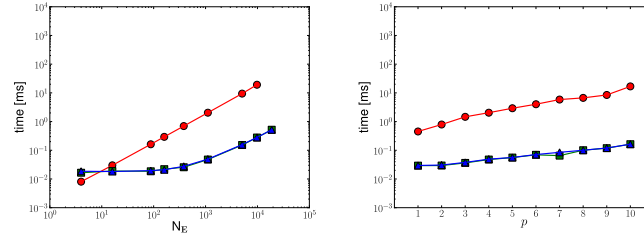
(a) Average kernel execution time.



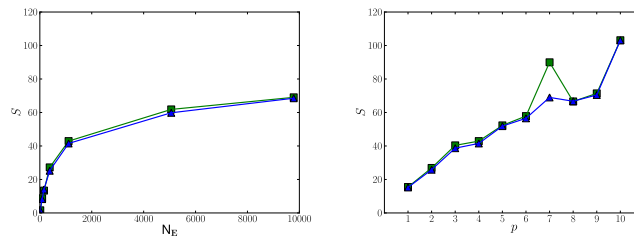
(b) Speedup.

Figure C.4: `mappingToElem` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , $P_{GPUBLAS}$.

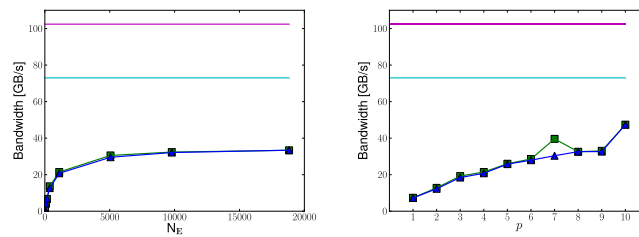
C.3.3 Evaluating the Physics



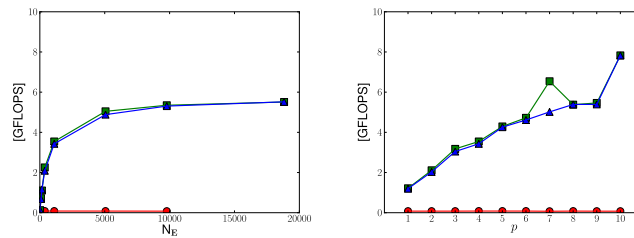
(a) Average kernel execution time.



(b) Speedup.

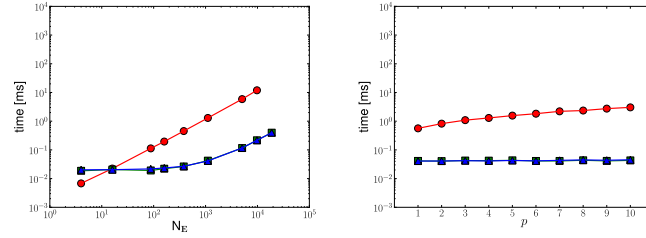


(c) Bandwidth comparison. Theoretical and practical bandwidth.

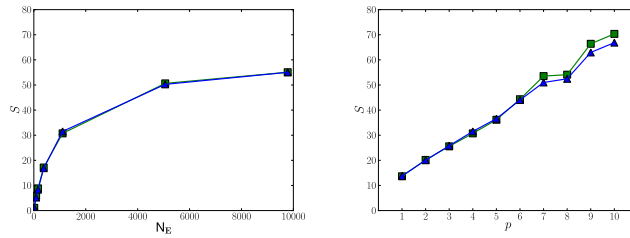


(d) Floating point operations.

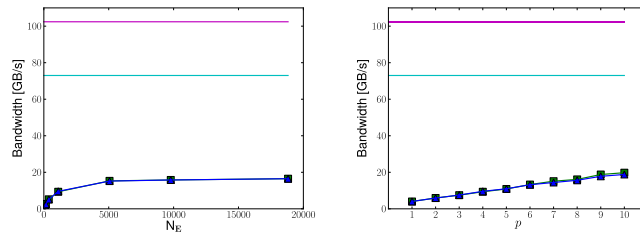
Figure C.5: `evaluate_sf` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , $P_{GPUBLAS}$.



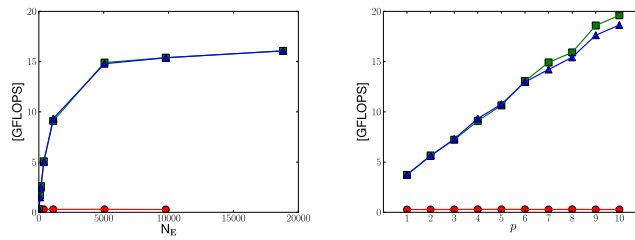
(a) Average kernel execution time.



(b) Speedup.



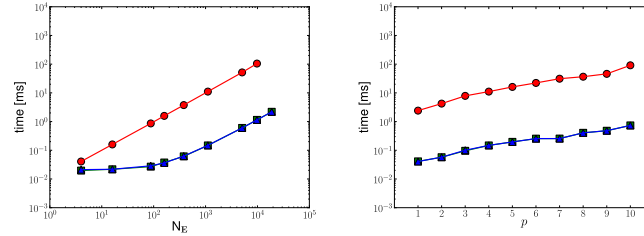
(c) Bandwidth comparison. Theoretical and practical bandwidth.



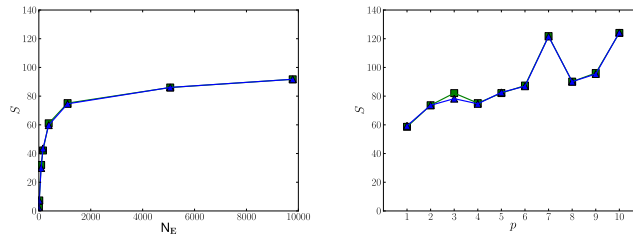
(d) Floating point operations.

Figure C.6: `evaluate_q` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , $P_{GPUBLAS}$.

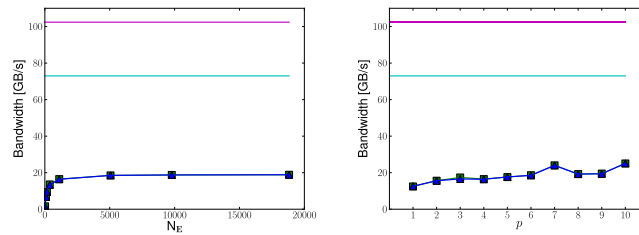
C.3.4 Multiplying by the Jacobians



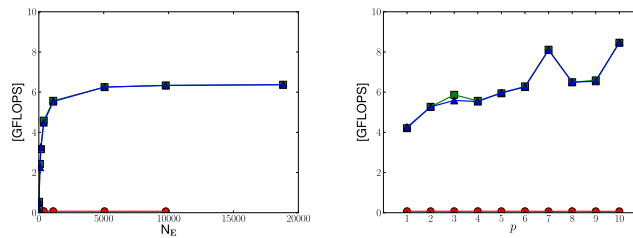
(a) Average kernel execution time.



(b) Speedup.

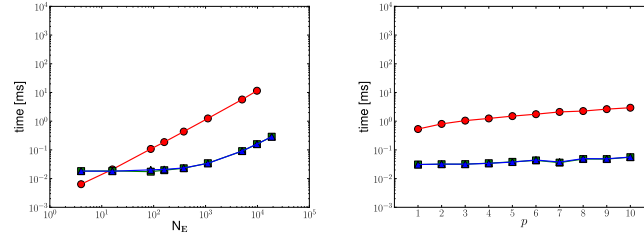


(c) Bandwidth comparison. Theoretical and practical bandwidth.

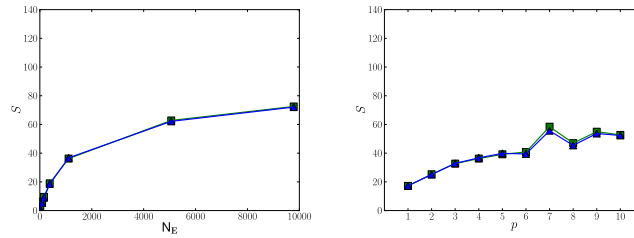


(d) Floating point operations.

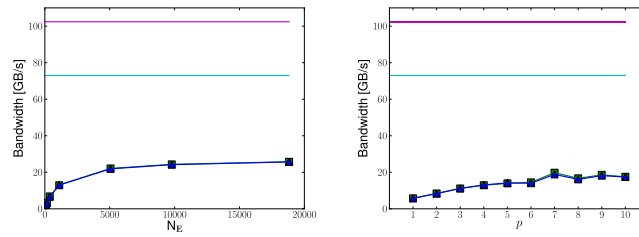
Figure C.7: `jacobian_sf` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , P_{GPUBLAS} .



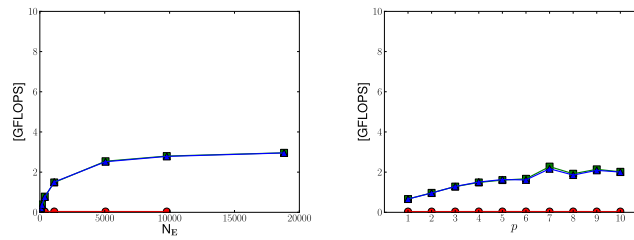
(a) Average kernel execution time.



(b) Speedup.



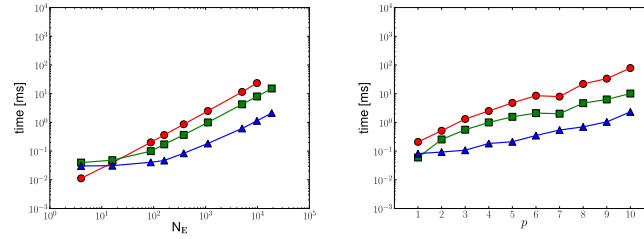
(c) Bandwidth comparison. Theoretical and practical bandwidth.



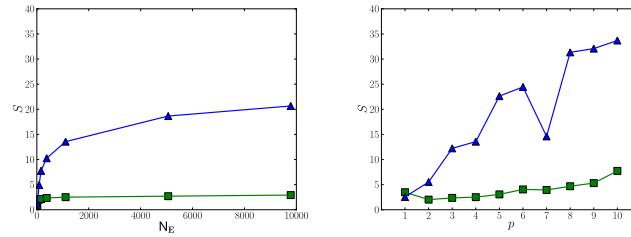
(d) Floating point operations.

Figure C.8: `jacobian_q` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , P_{GPUBLAS} .

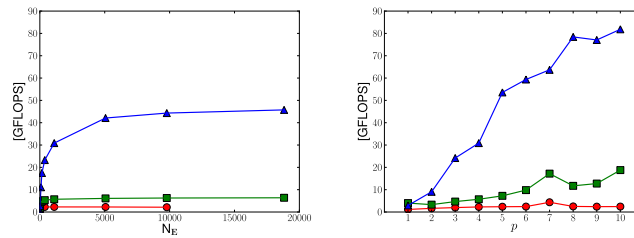
C.3.5 Matrix-matrix Products



(a) Average kernel execution time.

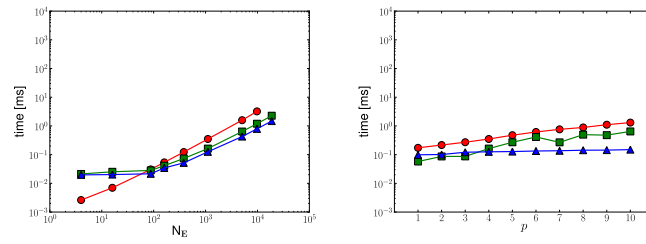


(b) Speedup.

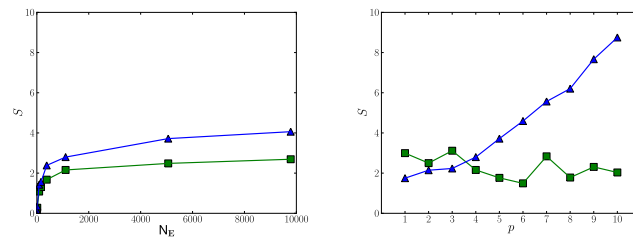


(c) Floating point operations.

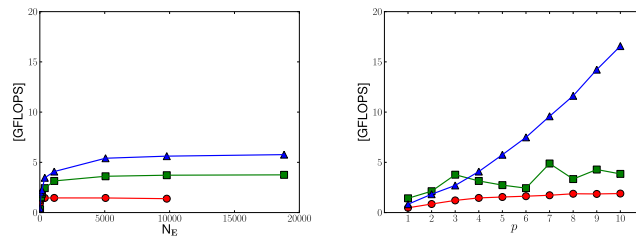
Figure C.9: `collocationU` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , $P_{GPUBLAS}$.



(a) Average kernel execution time.

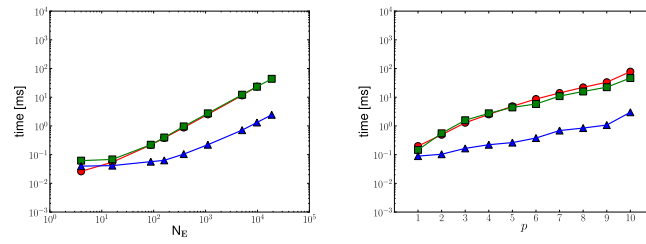


(b) Speedup.

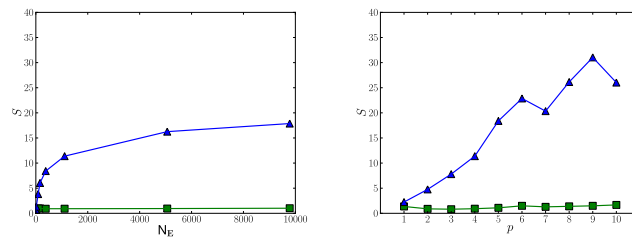


(c) Floating point operations.

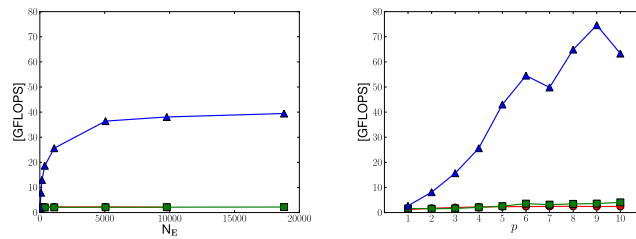
Figure C.10: `collocationUF` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , $P_{GPUBLAS}$.



(a) Average kernel execution time.

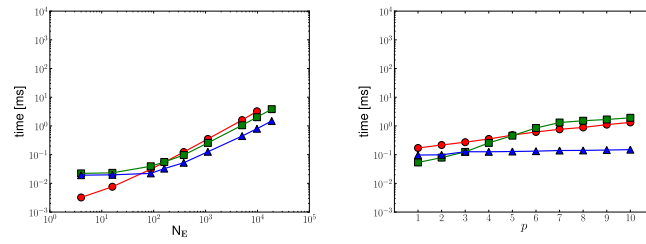


(b) Speedup.

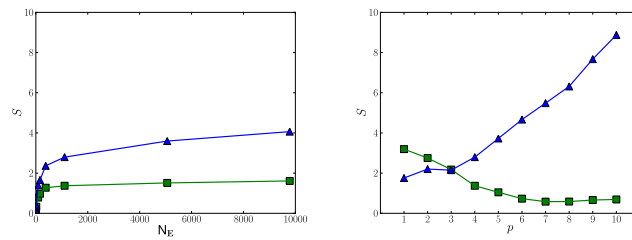


(c) Floating point operations.

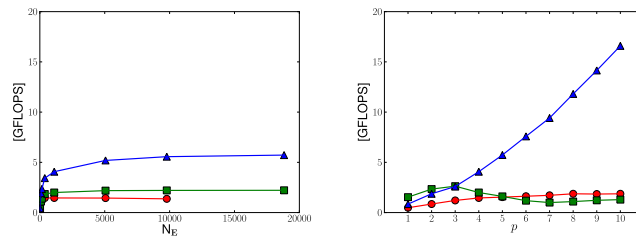
Figure C.11: `gemm_sf` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , $P_{GPUBLAS}$.



(a) Average kernel execution time.



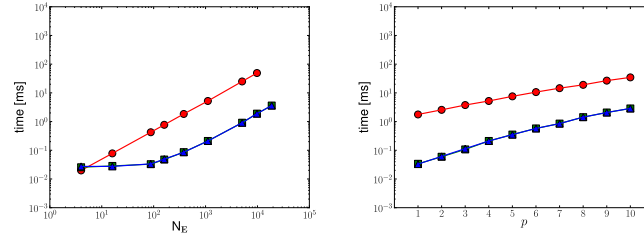
(b) Speedup.



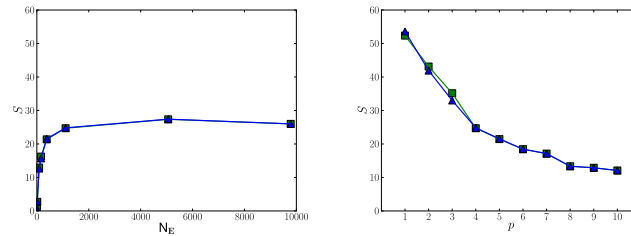
(c) Floating point operations.

Figure C.12: `gemm_q` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , $P_{GPUBLAS}$.

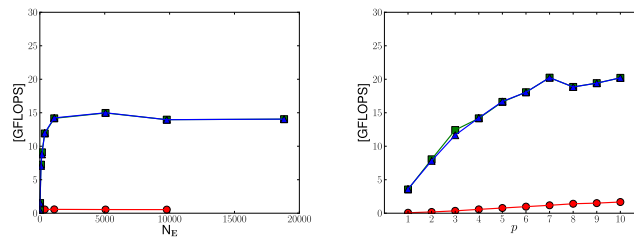
C.3.6 Solving the Linear System



(a) Average kernel execution time.



(b) Speedup.



(c) Floating point operations.

Figure C.13: `solve` kernel performance as a function of N_E (left column, $p = 4$) and as a function of p (right column, $N_E = 1108$) for P_{CPU} , P_{GPU} , P_{GPUBLAS} .

Appendix D

Computer characteristics

D.1 CPU Characteristics

The computer used for the CPU calculations, `gameboy`, has eight processors, each an Intel(R) Xeon(R) CPU X5550. These processors have

- a 2.67 GHz clock rate;
- four CPU cores;
- eight threads;
- and a 8 MB cache.

D.2 GPU Characteristics

The GPUs provided by the Institute of Mechanics, Materials and Civil Engineering are four Tesla C1060. Their characteristics are¹ presented in Table D.1.

¹Output from the `deviceQuery.cu` provided with NVIDIA's SDK.

Device 0: “Tesla C1060”	
CUDA Driver Version:	3.0
CUDA Runtime Version:	3.0
CUDA Capability Major revision number:	1
CUDA Capability Minor revision number:	3
Total amount of global memory:	4294770688 bytes
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:	2147483647 bytes
Texture alignment:	256 bytes
Clock rate:	1.30 GHz
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)

Table D.1: GPU characteristics.

The output from the bandwidth test (`bandwidthTest.cu` kernel in the SDK) is

```
./bandwidthTest Starting...
Running on...
Device 0: Tesla C1060
Quick Mode
Host to Device Bandwidth, 1 Device(s), Paged memory
  Transfer Size (Bytes)      Bandwidth(MB/s)
  33554432                   4795.8
Device to Host Bandwidth, 1 Device(s), Paged memory
  Transfer Size (Bytes)      Bandwidth(MB/s)
  33554432                   2995.3
Device to Device Bandwidth, 1 Device(s)
  Transfer Size (Bytes)      Bandwidth(MB/s)
  33554432                   73392.1
[bandwidthTest] - Test results:
PASSED
Press <Enter> to Quit...
-----
```


Appendix E

Notations

Notation	Significations	index	units
N_s	number of nodes on an element	i	-
M_s	number of nodes on a face	j	-
N_T	number of faces per element	-	-
N_E	number of elements	e	-
M_T	number of interfaces	t	-
N_G	number of integration points per element	g	-
M_G	number of integration points per face	g	-
D	number of coordinates in element reference space	α	-
D_F	number of coordinates in face reference space	α	-
N_F	number of unknown fields	fc	-
\mathcal{O}	occupancy	-	%
\mathcal{F}	floating point operations	-	flops
\mathcal{K}_o	ratio of kernel overhead to kernel execution time	-	%
B_{theo}	theoretical bandwidth	-	GB/s
B_{pract}	practical bandwidth	-	GB/s
P_{CPU}	program running on the CPU	-	-
P_{GPU}	program running on the GPU	-	-
P_{GPUBLAS}	program running on the GPU with the CUBLAS library	-	-

Table E.1: Symbols and their significance.