Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Laurea Magistrale in Ingegneria Matematica

A proposal for the implementation of high order discontinuous Galerkin methods for elliptic problems on GPUs

Autore:
Paolo Gorlani
797869

Relatore:
Prof. L. Bonaventura
Correlatrice:
Prof.ssa P. F. Antonietti

A.A. 2014-15

Contents

List of Figures

List of Tables

Introduction

This thesis presents a proposal for the implementation on graphics processing units (GPUs) of Discontinuous Galerkin methods for elliptic problems.

The technology of graphics processing units has shown significant performance and energy efficiency for many scientific applications. High performance computers based on GPUs are crucial in large-scale modelling. A total of 69 systems on the TOP500 list [1] of November 2015 are using GPUs.

The graphics processing units are not stand-alone devices. They are included in a computer system which contains at least a central processing unit (CPU). The GPUs provide more floating point operations per second and a wider memory bandwidth than CPUs.

There are drawbacks, however. The software development heavily depends on the architecture of the device. The GPUs are less versatile than the CPUs. A CPU can run a limited amount of independent threads at the same time (usually, at most 32). Every operation is independently executed from the others. A GPU can run a massive number of operations at the same time (approximately 10^8), but in order to reach the peak performance, the instructions must be the same for all threads. Moreover, in order to employ completely the GPU memory bandwidth, memory accesses need to coalesce, i.e., threads must access sequential memory locations. The developer of GPUs code needs to consider these constraints.

The main performance bottlenecks for the implementation of Finite Element methods on GPUs seem to be the assembly of the global stiffness matrix and the sparse matrix vector product [13]. The implementation of those tasks must be rethought in order to exploit the full power of GPUs.

Articles [14] and [15] provide a general analysis of the assembly strategies of the Finite Element methods stiffness matrices on a GPU. These strategies are quite general, since they can be applied to a wide class of Finite Element methods on unstructured meshes. As a consequence, they are not easy to implement in an optimal fashion on GPUs.

In order to simplify the code development, in this thesis, we have focused exclusively on Discontinuous Galerkin methods on structured quadrilateral meshes. The stiffness matrices of the Discontinuous Galerkin methods are larger than those of other Finite Element methods. On the other hand, due to the fact that different elements do not share any degrees of freedom, the vectors of the degrees of freedom in Discontinuous Galerkin methods can be arranged in order to exploit easily the full potentials of GPUs.

Most of the articles related to the solution of Discontinuous Galerkin methods on GPU solve hyperbolic equations with explicit time discretization methods. As discussed in [12], the

[1]`http://www.top500.org/lists/2015/09/` The TOP500 ranks the 500 most powerful computer systems in the world.

implementation of such methods mostly involves matrix-matrix multiplications.

The goal of this thesis is to provide instead a strategy for the GPU implementation of Discontinuous Galerkin methods for the numerical solution of elliptic problems. The discretization of elliptic methods, as well as the implicit time discretization of hyperbolic or parabolic problems, requires the solution of a system of linear equations, which is usually solved with an iterative method. The main operation in iterative methods is the matrix-vector product. We propose an implementation of the matrix vector product on GPU in which the stiffness matrix is not stored in memory, but rather recomputed at each iteration, in order to avoid the large latency of global memory transactions. In order to test the potentials of our proposal, we developed a code which solves the Poisson problem with Dirichlet boundary conditions using the Symmetric Interior Penalty Galerkin method with high order basis functions. The resulting method has proven to be effective in exploiting the special features of the GPU architecture and a preliminary multi-GPU implementation has shown potential for good parallel scaling.

This work has been partially carried out at the Centro Svizzero di Calcolo Scientifico (CSCS, Lugano, Switzerland). CSCS provided access to Piz Daint, a Cray XC30 system with a total of 5272 compute nodes, each of them equipped with an NVIDIA Tesla K20X. This enabled us to develop and test a multi-GPU version of our Poisson problem solver.

Chapter 1 introduces the fundamental concepts of the general-purpose computing on GPUs along with the available parallel computing platforms and the programming model of CUDA, which is the NVIDIA programming environment used in the development of our code. Chapter 2 summarizes the main theoretical results of the Symmetric Interior Penalty Galerkin method for diffusion-reaction problems. Moreover, it introduces the formulation of the basis functions employed by the Discontinuous Galerkin method we chose to implement. Chapter 3 provides a description of the proposal for the implementation of the matrix vector product. A brief description of the code is presented. Chapter 4 is entirely devoted to the validation and the performance measures of the code produced.

We acknowledge the help and support in our development of William Sawyer, computational scientist at CSCS, who has shown his interest for this project and has provided useful advices and Mauro Bianco, computational scientist at CSCS, who has developed and has supplied the `gcl` library used in the multi-GPU version of the code.

Chapter 1

General-purpose computing on graphics processing units

Graphics Processing Units were born in order to accelerate the creation of the images displayed on a screen. They are present in most of today's electronic devices, from high-end computers to cheap smartphones.

The Graphics Processing Unit term was born in the late 1990's in order to describe *graphics cards* best suited for the execution of three-dimensional video games. Three-dimensional computer graphics involves the memory-intensive work of texture mapping and the computation-intensive work of rotation of the polygons. These tasks require large memory transactions and a massive number of integer and floating-point operations. These operations must be executed very fast in order to smoothly render the images in real-time. The Graphics Processing Units have a massive parallel architecture. They execute thousands of operations at the same time. They can provide more floating point operations per second than the CPUs. Figure 1.4 shows floating point performance in single and double precision of NVIDIA GPUs and Intel CPUs. In particular, NVIDIA Geforce GPUs, designed for video games, show their supremacy on single precision operations. On other hand, NVIDIA Tesla GPUs, devoted to calculus, are the best in double precision. Furthermore, GPUs greatly outpace CPUs in memory bandwidth; Figure 1.5 shows the time evolution of the memory bandwidth of NVIDIA GPUs and Intel CPUs.

Because graphics as a whole involves dense matrix and vector operations, engineers and scientists have studied the use of GPUs for generic calculations. The General-Purpose Computing on Graphics Processing Units is the usage of graphic cards to perform calculations that are not related to graphics. This trend started in the early 2000's. At the beginning, scientific calculus developers adapted the native graphics programming interfaces. But this way was impractical. Such programming interfaces are best suited for the graphics, not for the generic computing.

The manufactures understood the emerging trend of general-purpose computing on graphics processing units. So, proper technologies and frameworks were developed to satisfy the demand.

NVIDIA created the Compute Unified Device Architecture (CUDA)[16], a parallel computing platform and a programming model. CUDA allows the usage of CUDA-enabled graphics cards for general purposes.

Nowadays, NVIDIA has produced GPUs without the video output which are completely devoted to scientific computing. Today these products equip the number 2 system [1] and the number 7 system [2] of the TOP500 list of November 2015.

The CUDA toolkit[18] is a collection of tools and mathematical libraries which creates a C\C++ development environment. Furthermore, the CUDA toolkit contains a lot of documentation about the development and the optimization of the code. So, we chose to develop our code in CUDA. In any case, we have made a minor usage of CUDA toolkit libraries in order to easily port our code to other frameworks.

1.1 CUDA overview

A thread is a sequence of instructions that can be executed in parallel with others. CUDA has a Single Instruction Multiple Threads architecture. This is not a real computational class. It is much more a commercial name which describes NVIDIA technology. This is a versatile architecture belonging to the Single Instruction Multiple Data class of the Flynn's taxonomy. SIMD includes computers in which all threads perform the *same operation* on *multiple data points* simultaneously, i.e. all the threads need to execute the same instruction. SIMT relaxes this assumption, it enables developers to write code for independent threads.

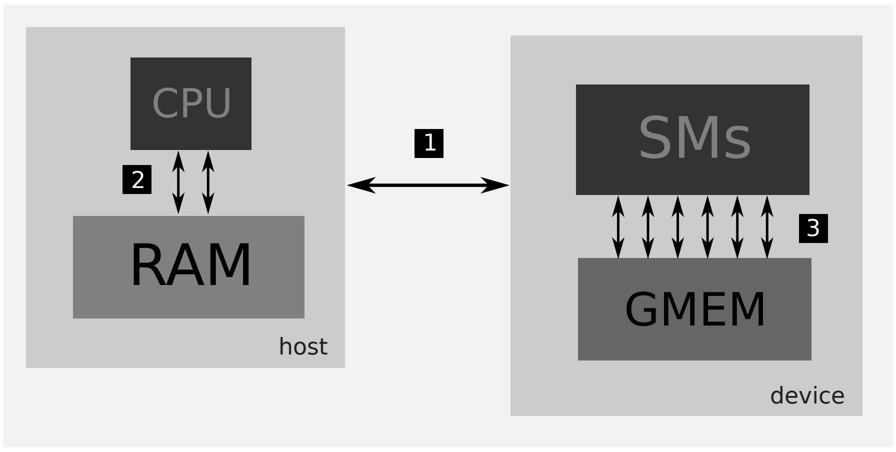


Figure 1.1: Host and device scheme.

Usually a GPU is seen as a *device* connected to a *host* system (computer) by a *bus* (e.g. PCIe). Figure 1.1 shows the host and device connections scheme. Bus [1] connects the device and the host. Buses [2] and [3] connect host and device processing units to their main memory. The indicative memory bandwidth are: [1] 15 GB/s (for PCIe x16 3.0), [2] 34 GB/s (for DDR4-2133) and [3] 250 GB/s (for Tesla K20x) One of the key characteristics of GPUs is the high bandwidth between processing units and main memory. The bus connecting the CPU to the GPU [1] is the bottleneck of the system. So, it is important to minimize the data transfers between the host and the device.

[1]Titan (DOE/SC/Oak Ridge National Laboratory) uses NVIDIA K20x
[2]Piz Daint (Swiss National Supercomputing Centre) uses NVIDIA K20x

The host and the device are two separate entities; each one has its own memory space and computational power. In this thesis,

- the *CPU* is the host processing unit, the *RAM* is the host main memory;
- *Streaming Multiprocessors* (*SMs*) are the device processing units, the *global memory* (*GMEM*) is the device main memory.

The RAM and the device global memory are made of the same type of memory integrated circuits, DDR SDRAM. They share *almost* the same technological characteristics. The big difference between the host and the device main memorie comes from the number of interfaces connecting them to their processing units. The bus connecting the CPU and the RAM has two interfaces. Usually, there are more interfaces between the Streaming Multiprocessors and the GMEM (e.g. 6 in Tesla K20x). The bus of the device can carry more data than the bus of the host. The width of an interface is 64 bit. So, 128 bit per clock are transmitted between the CPU and the RAM, 384 bit per clock between the Streaming Multiprocessors and the global memory.

Many computing architectures have been released since the introduction of CUDA. Compute capability (cc) numbers distinguish them; they are defined by

- major revision number, it describes the core architecture - 3 for devices based on the Kepler architecture, 2 for devices based on the Fermi architecture and 1 for devices based on the Tesla architecture;
- minor revision number, an improvement to core architecture.

For instance, compute capability of Tesla K20x is 3.5. Each architecture features different characteristics, while the programming model is the same.

1.2 Programming model

In CUDA, the code is written in a thread-wise way. Kernels are functions runned on the device, called by the host. Their declaration is preceded by the `__global__` keyword. The kernels cannot return anything, so they are declared as `void`. They accept any type of arguments, but pointers must refer to device global memory addresses.

1.2.1 Thread, block and grid

The kernels are executed by a chosen number of threads on the device. Threads are organized in blocks, forming a grid. This organization is reflexed in a system of indexes which assigns specific data to every thread.

Listing 1.1: Kernel code.

```
__global__ void array_sum(int n, double * a, double * b, double * c)
{

  int idx = blockDim.x*blockIdx.x + threadIdx.x;

  if(idx<n)
    c[idx] = a[idx] + b[idx];

}
```

For instance, Listing 1.1 shows a kernel which computes the sum of two vectors. Every value of the vector index is assigned to a thread. The variable `threadIdx.x` is the index-in-the-block of the thread, the variable `blockIdx.x` is the block index, the variable `blockDim.x` contains the grid dimension. So, `idx` is the global index of the element processed by the thread. At the end, each thread does a sum of two elements with the same index and writes the result in the corresponding location.

The previous kernel is launched by the following host code.

Listing 1.2: Host code.

```
#include<iostream>
#include"array_sum_kernel.h"


int main()
{

  const int array_size = 100;

  int threads_per_block = 32;
  int grid_size = (array_size + threads_per_block - 1 )/threads_per_block;
  std::cout<<grid_size<<std::endl;

  // declaration of 3 arrays on host
// host_x pointers contain host memory addresses
double * host_a = new double[array_size];
double * host_b = new double[array_size];
double * host_c = new double[array_size];

// filling of two arrays on host
for(int i = 0; i<array_size; ++i)
{
  host_a[i] = double(i)*.745;
  host_b[i] = double(i)*.356;
}

// pointers created to store device addresses
double * dev_a;
double * dev_b;
double * dev_c;

// device memory allocation
cudaMalloc(&dev_a, array_size*sizeof(double));
cudaMalloc(&dev_b, array_size*sizeof(double));
cudaMalloc(&dev_c, array_size*sizeof(double));

// NOW dev_x contains device addresses

// COPY host array TO device
cudaMemcpy(dev_a, host_a, array_size*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, host_b, array_size*sizeof(double), cudaMemcpyHostToDevice);

// launch the kernel
array_sum<<< grid_size, threads_per_block >>>(array_size, dev_a, dev_b,
    dev_c);

// COPY device results TO host
cudaMemcpy(host_c, dev_c, array_size*sizeof(double), cudaMemcpyDeviceToHost);

// show results
for(int i = 0; i<array_size; ++i)
  std::cout<<host_c[i]<<" ";
std::cout<<std::endl;

// free device memory
cudaFree(dev_a);
cudaFree(dev_b);
cudaFree(dev_c);

// free host memory
delete[] host_a;
delete[] host_b;
delete[] host_c;
```

```
63
64   return 0;
65
66 }
```

A kernel works only on data already present on the device. The data needs to be transferred from the host to the device before launching the kernel. So, in order to execute such kernel, Listing 1.2:

1. allocates device memory (lines 33-35);
2. transfers vector data from the host to the device (lines 40-41);
3. launches the kernel (line 44);
4. copies the result from the device to the host (line 47).

The above points represents the basic structure of every CUDA program.

1.2.2 Memory hierarchy

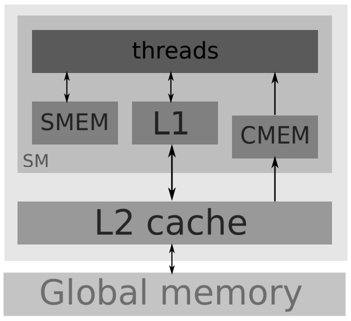


Figure 1.2: Memory hierarchy diagram. SMEM: shared memory. CMEM: constant memory. Registers are not shown.

There are different layers of memory between threads and the global memory. Below, they are described going from the *fastest* to the *slowest*:

- the *registers* are the working memory of the thread: they store all the variables which are not owned by the following types;
- the *constant memory* is a read-only memory: when all the threads of a warp read the same memory location it is as fast as the registers;
- the *shared memory* is in common between all the threads in a block; this memory creates a sort of communication between the threads in the same block; shared memory is a lot faster compared to global memory;
- the *local memory* is used when a thread runs out of all the available registers (register spilling). The local memory could be stored on the global memory. So, the local memory access might be very slow.

Caches L1 and L2 are present in compute capability 2.0 and above. The programmer can use both for cache global memory access or L1 can be used to store registers only, limiting the usage of global memory in case of register spilling.

1.3 How the computation is performed

A *Streaming Multiprocessor* consists of a number of central processing units linked together to enable parallel processing. Figure 1.3 shows the functional units of a Streaming Multiprocessor; the *Core* blocks perform logic and arithmetic computations, each of them contain an integer arithmetic logic unit and a floating point unit; the *LD/ST* blocks are the load and store units which manage the memory transactions; the *SFU* blocks are the special function units which execute transcendental instructions such as sin, cosine, reciprocal and square root. A device usually has many Streaming Multiprocessors (e.g. 14 in Tesla K20x), they work independently from each other.

The blocks, that run a *kernel*, are divided in warps (i.e. 32 threads). The Streaming Multiprocessor creates, manages, schedules, and executes the warps. The instructions are executed simultaneously on a warp.

The *latency* is the number of clock cycles required to complete the execution of an instruction. Different instructions have different latency. For instance, a global memory transaction has a latency of 400-800 cycles, while register calls have a latency of only 24 cycles. Arithmetic instructions have also a latency. For instance, the latency of double precision instructions are higher than that of single precision instructions.

The *throughput* is the ratio between data transferred or instructions executed and the time elapsed. In order to increase performance, the throughput needs to be maximized and the latency needs to be minimized.

Streaming Multiprocessors are able to change context (registers, instruction counters ...) between active warps with no effort. So, if an instruction of a warp requires many latency cycles, this latency can be hidden by running an instruction from another active warp.

Traditionally, CPUs try to increase performance minimizing the latency. In order to achieve this goal, the CPU are designed to improve clock frequency and to use complex instruction sets. On the other hand, Streaming Multiprocessors achieve high performance hiding the latency. The GPU goal is to run a massive number of threads in parallel.

The parallelism is realized in two ways:

- the thread level parallelism (TLP) is the execution of many independent threads in order to maximize the *occupancy* (see Section 1.4.1), i.e. the ratio between the actual active threads on the maximum number of active threads. The effective occupancy is important, but it is not the ultimate index of performance. Sometimes, a low occupancy could be enough to hide latency, see e.g. [21].

- the instruction level parallelism (ILP) is the execution of independent instructions on the same thread. ILP hides latency with a lower occupancy.

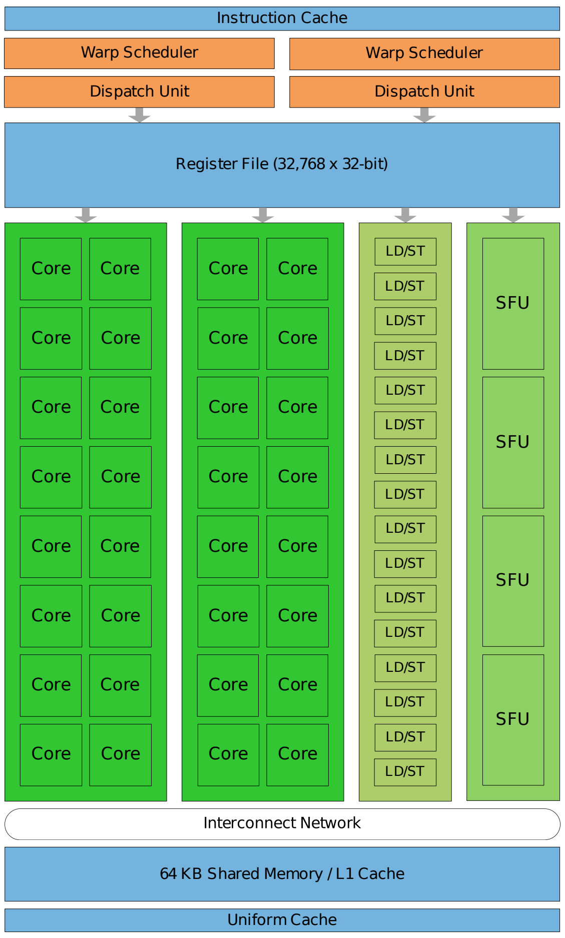


Figure 1.3: Streaming Multiprocessor scheme (cc 2.0). Figure taken from the Fermi Architecture white paper.

1.4 How to achieve good performance

Every task performed by a GPU can be developed much more easily on a CPU. The usage of the GPU needs to bring performance improvements in order to be useful.

The main causes of GPU bad performance are:

- *Non coalescence* of global memory access. A global memory transition is said to coalesce, if caches are fully employed. In this way, the maximum memory bandwidth is reached. The concept of coalescence depends on the compute capability of the device. But, in general, there is coalescence if all the threads in the same warp access sequential memory locations.

- *Banks conflicts* in shared memory. The shared memory is divided into banks, they can be accessed simultaneously by different threads. But, if different threads read or write data in the same memory bank, there is a bank conflict and the transitions are serialized.

- *Threads divergence* occurs when threads belonging to the same warp execute different instructions. The main cause of divergence is conditional programming (loops with different iteration numbers or use of `if` statements that behave differently on different threads). SIMT architecture allows the execution of divergent threads, but in such cases the instruction are not executed in parallel anymore, the performance decreases dramatically because the instruction executions are serialized.

For instance, the summation of two vectors is *bandwidth bounded* because this task is memory intensive; floating point operations are minimum. So, in this case is fundamental to have coalescence in memory access.

On other hand, there are task which are *computation bounded.* In them the number of arithmetic instructions are higher than memory transaction. In such case, it is essential to avoid threads divergence.

1.4.1 Occupancy

A way to evaluate the performance is consider the occupancy, which is the percentage of the Streaming Multiprocessor usage defined as

$$occupancy = \frac{\#\ of\ active\ warps}{max\ \#\ of\ active\ warps}$$

An *active warp* is a warp actually executed by a Streaming Multiprocessor. The maximum number of active warps on a Streaming Multiprocessor depends upon the compute capability of the device, see Table 1.1.

Compute capability	1.3	2.x	3.x	5.x
Max active warps	32	48	64	64

Table 1.1: Maximum number of active warps.

In order to maximize the occupancy, we need to keep in mind three device constrains:

- *Number of blocks.* A Streaming Multiprocessor can run only a relatively small number of blocks simultaneously. If the blocks of a kernel contain too few threads, the Streaming Multiprocessor cannot activate enough warps. Therefore, the block size needs to have an adequate size; it must be less than the maximum number of active threads but big enough to occupy the Streaming Multiprocessor.
- *Registers.* The number of registers in a Streaming Multiprocessor is limited. The register usage is proportional to the program complexity. Code complexity increases the number of registers used. Unfortunately, the developer cannot choose directly how use the registers, since they are assigned by the compiler. At most, the developer can limit the usage of the registers with a compiler flag. But this can be counterproductive, because the exceeding data will be stored in the local memory, which is slower than the registers. In order to lower register usage, it is important to maintain the code simple. For this purpose , it can be helpful to split the work on multiple kernels.
- *Amount of shared memory.* The shared memory is limited as well. If the blocks use too much shared memory, it is not possible to run simultaneously more of them.

If a kernel uses too many registers or shared memory, it cannot reach a full occupancy. The usage of these resources influences the choice of the block size.

Compute capability	1.3	2.x	3.x	5.x
Maximum active block	8	8	16	32
Registers	16K	32K	64K	64K
Shared memory	16K	48K	48K	64K-96K

Table 1.2: Streaming multiprocessor available resources.

For example, in compute capability 2.1, a Streaming Multiprocessor has 32768 registers available. A kernel can reach full occupancy only if every thread uses, at most, 21 registers. The block size needs to be a multiple of 32 (the warp size) and must be a divisor of the maximum number of active threads (in this case 48×32). The active blocks must be at maximum 8, and each one can use at the most 49152 bytes of shared memory.

Finally, it is to be remembered that a high occupancy is desirable, but is not the ultimate index of performance. A lower occupancy could be enough to hide latency, see e.g. [21].

1.5 Characteristics of GPU hardware

Table 1.3 shows some examples of GPUs. They have been employed by the author during the development and the testing of the code prepared for this thesis. Moreover, preliminary performance tests had been made using Tesla M2050 and Tesla K520; both are available thought the Amazon cloud service, AWS. The Centro Svizzero di Calcolo Scientifico allowed the author to use the Piz Daint system, which is equipped with Tesla K20x. The performances results, which are provided in the next chapters, have been achieved using the Tesla K20x of the Piz Daint system.

	CUDA capability	# of Streaming Multiprocessor	Max # of register (32 bit) per thread	Category
GeForce GT 610	2.1	1	63	Low-end graphics card
Tesla C1060	1.3	30	128	Scientific computing (MOX)
Tesla M2050	2.0	14	63	Scientific computing (AWS)
Tesla K520	3.0	8	255	Scientific computing (AWS)
Tesla K20x	3.5	14	255	Scientific computing (CSCS)

Table 1.3: The characteristics of devices.

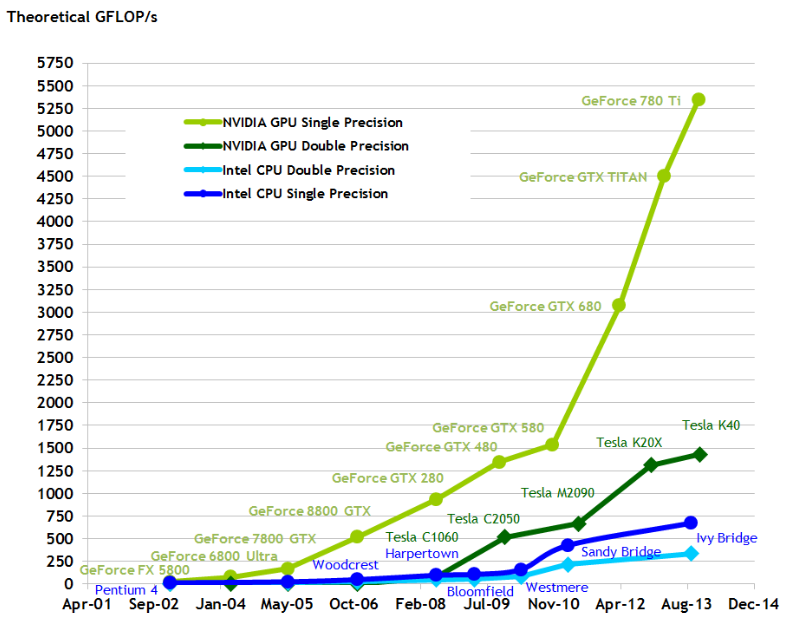


Figure 1.4: Floating-Point Operations per Second for the CPU and GPU. Figure taken from the CUDA Toolkit documentation.

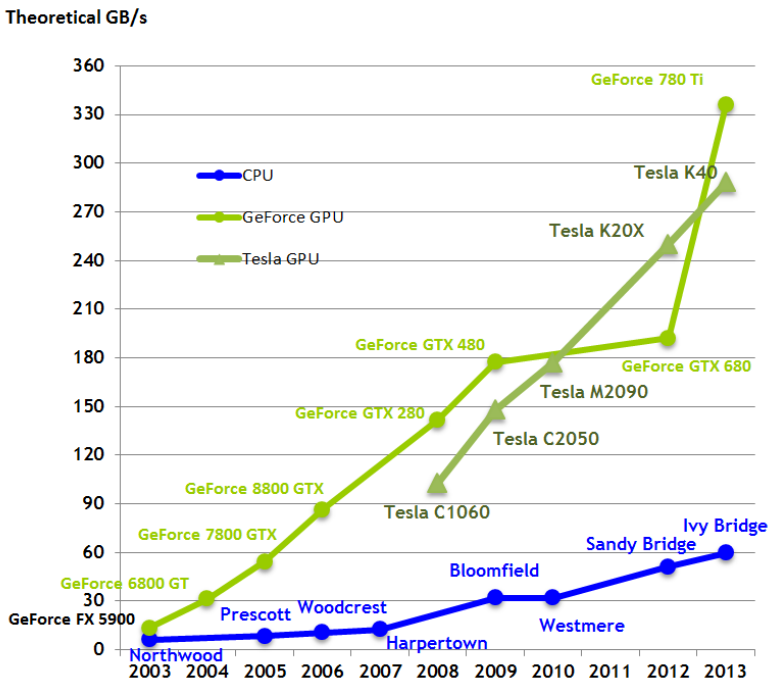


Figure 1.5: Memory Bandwidth for the CPU and GPU. Figure taken from the CUDA Toolkit documentation.

Chapter 2

Discontinuous Galerkin approximation of diffusion-reaction problems

2.1 Mathematical problem

In this thesis, we consider the following *boundary value problem* representing a simple diffusion-reaction problem:

Problem 2.1. *Given a limited domain $\Omega \subset \mathbb{R}^2$; the functions $\lambda, f : \Omega \to \mathbb{R}$ and $g : \partial\Omega \to \mathbb{R}$; find the solution $u : \Omega \to \mathbb{R}$ that satisfies:*

$$\begin{aligned} -\Delta u + \lambda u &= f \ \textit{in } \Omega \\ u &= g \ \textit{in } \partial\Omega \end{aligned} \tag{2.1}$$

where the function g is the Dirichlet boundary datum.

2.2 Preliminary Concepts

2.2.1 Weak formulation

In order to ease the presentation, we consider $g = 0$. Let us to define V as

$$V \ \equiv \ H_0^1(\Omega) \ = \ \{ \ v \in H^1(\Omega) : \ v|_{\partial\Omega} = 0 \ \},$$

where

$$H^1(\Omega) \ = \ \{ \ v \in L^2(\Omega) : \ \nabla v \in [L^2(\Omega)]^2 \ \}.$$

Let us to multiply (2.1) by an arbitrary test function $v \in V$. Then, we integrate by parts the Laplacian term using the *Gauss Green formula*, so we obtain the weak form of Problem 2.1.

Problem 2.2. *Weak formulation.*

$$\textit{find} \quad u \in V \quad \textit{such that} \quad \int_\Omega (\nabla u \nabla v + \lambda u v) = \int_\Omega f v \quad \forall v \in V$$

On other hand, if $g \neq 0$, we introduce $R_g \in H^1(\Omega)$ such that $R_g|_{\partial\Omega} = g$. Then $u = \mathring{u} + R_g$, where $\mathring{u} \in V$ is the solution of

$$\int_\Omega (\nabla \mathring{u} \nabla v + \lambda \mathring{u} v) = \int_\Omega f v - \int_\Omega (\nabla R_g \nabla v + \lambda R_g v) \qquad \forall v \in V \ .$$

Let $a : V \times V \to \mathbb{R}$ be the bilinear form

$$a(u, v) = \int_\Omega (\nabla u \nabla v + \lambda u v) \ , \tag{2.2}$$

and let $F : V \to \mathbb{R}$ be the linear functional

$$F(v) = \int_\Omega f v - \int_\Omega (\nabla R_g \nabla v + \lambda R_g v) \ . \tag{2.3}$$

Problem 2.1 can be rewritten as follows.

Problem 2.3. *Variational formulation.*

$$\textit{find} \quad u \in V \quad \textit{such that} \quad a(u, v) = F(v) \quad \forall v \in \ V$$

2.2.2 Well-posedness of the variational formulation

The Lax-Milgram theorem guarantees the existence, the uniqueness and the stability of the solution of Problem 2.3.

Theorem 2.4. *Let V be an Hilbert space with a norm $||\cdot||_V$ and let $a : V \times V \to \mathbb{R}$ be a bounded bilinear form which is coercive, i.e.*

$$\exists \alpha > 0 : \quad a(v, v) \geq \alpha ||v||_V^2 \quad \forall v \in V \ .$$

Furthermore, let $F : V \to \mathbb{R}$ be a linear bounded functional. Then $\exists!\ u \in V$ solution of Problem 2.3 with the following estimate:

$$||u||_V \leq \frac{1}{\alpha} \sup_{v \in V \setminus \{0\}} \frac{|Fv|}{||v||_V} \ .$$

The hypotheses of the Lax-Milgram theorem are satisfied for Problem 2.3 by assuming suitable regularity for the data. Indeed, the bilinear form $a(\cdot, \cdot)$ is *bounded* and *coercive* if

$$\lambda \in L^\infty(\Omega) \quad \text{and} \quad \lambda > 0 \text{ a.e. on } \Omega \ .$$

The functional $F(\cdot)$ is *bounded* if

$$f \in H^{-1}(\Omega) \quad \text{and} \quad g \in H^{1/2}(\partial\Omega) \ .$$

Furthermore, if Ω is convex and Lipschitz, assuming instead

$$f \in L^2(\Omega) \quad \text{and} \quad g \in H^{3/2}(\partial\Omega) \tag{2.4}$$

leads to $u \in H^2(\Omega) \cap H_0^1(\Omega)$. Hypotheses (2.4) are fundamental to achieve *strong* solutions, soon this aspect will be very important.

2.2.3 Galerkin methods

The idea behind the Galerkin methods is to replace the space V by a *finite dimensional space* V_h defined on a mesh $\mathcal{K}_h$. Let $\Omega \subset \mathbb{R}^2$ be a bounded polygonal domain, a mesh $\mathcal{K}_h$ is a partition of Ω such that

$$\overline{\Omega} = \bigcup_{K \in \mathcal{K}_h} K ,$$

where $\overline{\Omega}$ is the closure of Ω. The parameter h represents the maximum diameter of $\mathcal{K}_h$ elements. In order to be suitable, V_h must satisfy the *approximability* property:

$$\lim_{h \to 0} \inf_{v_h \in V_h} ||v - v_h||_V = 0 \quad \forall v \in V .$$

The general form of the Galerkin method is the following:

Problem 2.5.

$$\textit{find} \quad u_h \in V_h \quad \textit{such that} \quad a_h(u_h, v_h) = F_h(v_h) \quad \forall v_h \in V_h$$

The bilinear form $a_h(\cdot,\cdot)$ and the linear functional $F_h(\cdot)$ are suitable approximation of $a(\cdot,\cdot)$ and $F(\cdot)$ from Problem 2.3.

We introduce the following two definitions.

Definition 2.6. *Problem 2.5 is said to be* conforming *if $V_h \subset V$, it is said to be* non-conforming *if $V_h \not\subset V$.*

In non-conforming problems, $a_h(\cdot,\cdot)$ cannot be defined on the entire V. So, it is important to find a subspace $V_* \subset V$ in which $a_h(\cdot,\cdot)$ is valid and such that $u \in V_*$, where u is the solution of Problem 2.3.

Let us to declare sum space

$$V(h) = V_* + V_h \tag{2.5}$$

such that the error $(u - u_h) \in V(h)$.

Definition 2.7. *Let $u \in V$ be the solution of Problem 2.3 and suppose that $a_h : V_h \times V_h$ can be extended to $V(h) \times V_h$. Problem* (2.5) *is said to be* strongly consistent *if the exact solution u solves it*

$$a_h(u, v_h) = F_h(v_h) \qquad \forall v_h \in V_h$$

If Problem 2.5 is strongly consistent, the so called *Galerkin orthogonality* holds true, i.e.

$$a_h(u - u_h, v_h) = 0 \quad \forall v_h \in V_h . \tag{2.6}$$

Strang Second Lemma provides an abstract error estimate for non-conforming Galerkin methods. In the case of a strongly consistent problem, it reads as follow.

Theorem 2.8. *Let u be the solution of Problem 2.3, which satisfies the hypotheses of Theorem 2.4. Let Problem 2.5 be a strongly consistent non-conforming approximation of Problem 2.3. Let $a_h : V_h \times V_h$ be a bilinear form which can be extended to $V(h) \times V_h$. Let $F_h : V_h \to \mathbb{R}$ be a linear bounded functional. The space $V(h)$ is endowed with $||\cdot||_{V(h)}$ such that $||\cdot||_{V_h} \leq ||\cdot||_{V(h)}$. The bilinear form $a_h(\cdot,\cdot)$ is bounded on $V(h) \times V_h$, i.e.*

$$\exists M_* : \quad |a_h(v, w_h)| \leq M_* ||v||_{V(h)} ||w_h||_{V_h} \quad \forall (v, w_h) \in V(h) \times V_h \ ,$$

and coercive on V_h, i.e.

$$\exists \alpha_* > 0 : \quad a_h(v_h, v_h) \geq \alpha_* ||v_h||^2_{V_h} \quad \forall v_h \in V_h \ .$$

Then $\exists!$ $u_h \in V$ solution of Problem 2.5 such that

$$||u_h||_{V_h} \leq \frac{1}{\alpha_*} \sup_{v_h \in V_h \setminus \{0\}} \frac{|F_h v_h|}{||v_h||_{V_h}} \ .$$

Moreover, the following estimate holds

$$||u - u_h||_{V_h} \leq \Big(1 + \frac{M_*}{\alpha_*}\Big) \inf_{w_h \in V_h} ||u - w_h||_{V(h)} \ .$$

2.3 Symmetric Interior Penalty Galerkin method

The Symmetric Interior Penalty Galerkin method is a *non-conforming* Galerkin method which belongs to the class of Discontinuous Galerkin methods.

2.3.1 Broken Sobolev spaces

Broken Sobolev spaces are the natural setting of Discontinuous Galerkin methods. If we have a mesh $\mathcal{K}_h$ on Ω, we define the *broken Sobolev spaces* $H^m(\mathcal{K}_h)$ as

$$H^m(\mathcal{K}_h) \ = \ \{ \ v \in L^2(\Omega) : \ v|_K \in H^m(K) \quad \forall K \in \mathcal{K}_h \ \} \ ,$$

where $m \geq 0$ is an integer. We endow $H^m(\mathcal{K}_h)$ with the norm

$$||v||_{H^m(\mathcal{K}_h)} \ = \ \Big[\ \sum_{K \in \mathcal{K}_h} ||v||^2_{H^m(K)} \ \Big]^{1/2} \ .$$

Clearly, it holds

$$H^m(\Omega) \ \subset \ H^m(\mathcal{K}_h) \ .$$

Moreover for $m = 0$, we simply write $L^2(\Omega)$ instead of $L^2(\mathcal{K}_h)$.

In this case, the space V_h is chosen as the finite dimensional *piecewise discontinuous polynomial space* defined as

$$\mathbb{P}^p(\mathcal{K}_h) \ = \ \{ \ v \in L^2(\Omega) : \ v|_K \in \mathbb{P}^p(K) \quad \forall K \in \mathcal{K}_h \ \} \ ,$$

where $\mathbb{P}^p(K)$ represents the space of polynomials of total degree p defined on K.

An important result, which we will use later, comes from the *discrete trace inequality*[4, p.28] that provides an upper bound on the face values for functions on $\mathbb{P}^p(\mathcal{K}_h)$.

Lemma 2.9. *There exists $C_{tr} > 0$ such that $\forall K \in \mathcal{K}_h$ and $\forall v \in \mathbb{P}^p(\mathcal{K}_h)$ it holds*

$$h_T^{1/2}||v||_{L^2(\partial K)} \leq C_{tr}||v||_{L^2(K)} \tag{2.7}$$

Where C_{tr} depends only on the mesh and on the polynomial degree p.

2.3.2 Jump and average operators

The set $\mathcal{F}_h$ contains the face of elements. In particular, $\mathcal{F}_h^i$ collects the interfaces between elements and $\mathcal{F}_h^b$ collects faces which are included in $\partial\Omega$.

$$\mathcal{F}_h = \mathcal{F}_h^i \cup \mathcal{F}_h^b$$

Let $K_1, K_2 \in \mathcal{K}_h$ be two contiguous element which share a face $F = K_1 \cap K_2 \in \mathcal{F}_h^i$. The outward normal vector $\mathbf{n}_F$ is oriented form K_1 to K_2. Let $v : \Omega \to \mathbb{R}$ be a regular enough function which admits a two-valued trace on $\mathcal{F}_h^i$. The averages of v on F are defined as

$$\{\{v\}\} = \tfrac{1}{2}(v|_{K_1} + v|_{K_2}) \ ,$$

and the jumps as

$$[\![v]\!] = (v|_{K_1} - v|_{K_2}) \ .$$

The averages and the jumps of v on $\mathcal{F}_h^b$ are conventionally defined as

$$[\![v]\!] = \{\{v\}\} = v \ .$$

2.3.3 Symmetric Interior Penalty Galerkin method formulation

The Symmetric Interior Penalty Galerkin method formulation for Problem 2.1 reads as follows.

Problem 2.10.

$$\textit{find} \quad u_h \in V_h \quad \textit{such that} \quad a_h(u_h, v_h) = F_h(v_h) \quad \forall v_h \in V_h$$

Where $a_h : V_h \times V_h \to \mathbb{R}$ and $F_h : V_h \to \mathbb{R}$ are equal to

$$\begin{aligned} a_h(u,v) &= \sum_{K \in \mathcal{K}_h} \int_K \nabla u \nabla v + \int_\Omega \lambda uv \\ &\quad - \sum_{F \in \mathcal{F}_h} \int_F \{\{\nabla u \cdot \mathbf{n}_F\}\}[\![v]\!] - \sum_{F \in \mathcal{F}_h} \int_F \{\{\nabla v \cdot \mathbf{n}_F\}\}[\![u]\!] \qquad (2.8) \\ &\quad + \sum_{F \in \mathcal{F}_h} \frac{\eta}{h_F} \int_F [\![u]\!][\![v]\!] \\ F_h(v) &= \int_\Omega fv - \sum_{F \in \partial\Omega} \int_F \Big(\nabla v \cdot n_F + \frac{\eta}{h_F} v\Big) g \qquad (2.9) \end{aligned}$$

The third, fourth and fifth term of (2.8) *are respectively called consistency, symmetry and penalty terms.*

In order to ease the analysis, we impose conditions 2.4 on the data and the domain. So, the exact solution of Problem 2.3 belongs to $H^2(\Omega)$. Furthermore,

$$V_h \; = \; \mathbb{P}^p(\mathcal{K}_h) \quad \text{and} \quad V(h) \; = \; H^2(\Omega) \; \cap \; \mathbb{P}^p(\mathcal{K}_h) \; .$$

Moreover, the following *bound on consistency term*[4, p.128] holds true.

Lemma 2.11. *For all $(v, w) \in V(h) \times V_h$:*

$$\Big| \sum_{F \in \mathcal{F}_h} \int_F \{\{\nabla v \cdot \mathbf{n}_F\}\} [\![w]\!] \Big| \le \Big(\sum_{T \in \mathcal{K}_h} \sum_{F \subset \partial T} h_F ||\nabla v \cdot \mathbf{n}_F||^2_{L^2(F)} \Big)^{1/2} |w|_J \tag{2.10}$$

We next show the coercivity of the bilinear form $a_h(\cdot,\cdot)$ usign the following energy norm

$$|||v|||_{ip} = \Big(||\nabla_h v||^2_{[L^2(\Omega)]^2} + ||\lambda^{1/2} v||^2_{L^2(\Omega)} + |v|^2_J \Big)^{1/2} ,$$

where $||\nabla_h v||_{[L^2(\Omega)]^d}$ is

$$||\nabla_h v||_{[L^2(\Omega)]^2} \; = \; \Big(\sum_{K \in \mathcal{K}_h} \int_K ||\nabla v||^2_{[L^2(K)]^2} \Big)^{1/2}$$

and $|v|_J$ is the jump semi-norm

$$|v|_J \; = \; \Big(\sum_{F \in \mathcal{F}_h} \frac{1}{h_F} ||[\![v]\!]||^2_{L^2(F)} \Big)^{1/2} \; ,$$

where the variable h_F is the diameter of the face F.

Problem 2.10 is well-posed if $a_h(\cdot,\cdot)$ is *coercive*, i.e.

$$\exists \alpha_* > 0 : \quad a_h(v_h, v_h) \ge \alpha_* |||v_h|||^2_{ip} \quad \forall v_h \in V_h$$

The coercivity is equivalent to the injectivity of the discrete linear operator representing a_h. In finite dimension, the injectivity is equivalent to the bijectivity: this makes the associated linear system solvable.

Lemma 2.12. *Coercivity[4, p.129]. For all $\eta > C^2_{tr}$ there exists $C_\eta > 0$ such that*

$$a_h(v_h, v_h) \ge C_\eta |||v_h|||^2_{ip} \qquad \forall v_h \in V_h \; .$$

Where $C_\eta = (\eta - C^2_{tr})(1+\eta)^{-1}$.

Proof. Using the trace inequality (2.7) and the fact that $h_F \le h_T$, we get

$$\sum_{T \in \mathcal{K}_h} \sum_{F \subset \partial T} h_F ||\nabla v \cdot \mathbf{n}_F||^2_{L^2(F)} \le \sum_{T \in \mathcal{K}_h} h_T ||\nabla v \cdot \mathbf{n}_F||^2_{L^2(\partial T)} \le C^2_{tr} ||\nabla_h v_h||^2_{[L^2(\Omega)]^2}$$

The above inequality in (2.10) leads to

$$\Big|\sum_{F\in\mathcal{F}_h}\int_F \{\{\nabla v_h\cdot\mathbf{n}_F\}\}[\![v_h]\!]\Big| \le C_{tr}||\nabla_h v_h||_{[L^2(\Omega)]^2}|v_h|_J$$
$$\le C_{tr}\Big[||\nabla_h v_h||^2_{[L^2(\Omega)]^2}+||\lambda^{1/2}v_h||^2_{L^2(\Omega)}\Big]^{1/2}|v_h|_J$$

Therefore,

$$\begin{aligned}a_h(v_h,v_h) \ \ge\ & ||\nabla_h v_h||^2_{[L^2(\Omega)]^2}+||\lambda^{1/2}v||^2_{L^2(\Omega)}\\ & -2C_{tr}\Big[||\nabla_h v_h||^2_{[L^2(\Omega)]^2}+||\lambda^{1/2}v_h||^2_{L^2(\Omega)}\Big]^{1/2}|v_h|_J\\ & +\eta|v_h|_J\end{aligned}$$

Let β be a positive number such that $\eta>\beta^2$, then for all $x,y\in\mathbb{R}$:

$$x^2-2\beta xy+\eta y^2 \ \ge\ \frac{\eta-\beta^2}{(1+\eta)}\ (x^2+y^2)$$

Applying this inequality with

$$\begin{aligned}x &= \Big[||\nabla_h v_h||^2_{[L^2(\Omega)]^2}+||\lambda^{1/2}v||^2_{L^2(\Omega)}\Big]^{1/2}\\ y &= |v_h|_J\\ \beta &= C_{tr}\end{aligned}$$

yields the coercivity of a_h with $C_\eta\ =\ (\eta-C_{tr}^2)\ (1+\eta)^{-1}$. □

We endow $V(h)$ with the following norm

$$|||v|||_{ip,*}=\Big(|||v|||^2_{ip}+\sum_{K\in\mathcal{K}_h}h_K||\nabla v\cdot n_K||^2_{L^2(\partial K)}\Big)^{1/2}$$

and prove the following result.

Lemma 2.13. *Boundedness[4, p.130]. There exists M, independent of h, such that:*

$$a_h(v,w_h)\le M|||v|||_{ip,*}|||w_h|||_{ip}\qquad \forall(v,w_h)\in V(h)\times V_h$$

Proof. Let $(v,w_h)\in V(h)\times V_h$.

$$\begin{aligned}a_h(v,w_h) \ =\ & \sum_{T\in\mathcal{K}_h}\int_T \nabla v\nabla w_h+\int_\Omega \lambda v w_h\\ & -\sum_{F\in\mathcal{F}_h}\int_F\{\{\nabla v\cdot\mathbf{n}_F\}\}[\![w_h]\!]\\ & -\sum_{F\in\mathcal{F}_h}\int_F\{\{\nabla w_h\cdot\mathbf{n}_F\}\}[\![v]\!]\\ & +\sum_{F\in\mathcal{F}_h}\frac{\eta}{h_F}\int_F[\![v]\!][\![w_h]\!] \ :=\ \mathcal{I}_1+\mathcal{I}_2+\mathcal{I}_3+\mathcal{I}_4+\mathcal{I}_5\end{aligned}$$

We can bound each of the terms on the right hand side as

$$
\begin{aligned}
&|\mathcal{I}_1| \leq ||\nabla_h v||_{[L^2(\Omega)]^d} ||\nabla_h w_h||_{[L^2(\Omega)]^d} \\
&|\mathcal{I}_2| \leq ||\lambda^{1/2} v||_{L^2(\Omega)} ||\lambda^{1/2} w_h||_{L^2(\Omega)} \\
&|\mathcal{I}_3| \leq \Big[\sum_{T \in \mathcal{K}_h} h_T ||\nabla v \cdot \mathbf{n}_T||^2_{L^2(\partial T)} \Big]^{1/2} |w_h|_J \leq |||v|||_{ip,*} |w_h|_J \\
&|\mathcal{I}_4| \leq C_{tr} |v|_J ||\nabla_h w_h||_{[L^2(\Omega)]^d} \\
&|\mathcal{I}_5| \leq \eta |v|_J |w_h|_J
\end{aligned}
$$

The proof is concluded collecting the above bounds. □

2.3.4 Error estimate and convergence

The Strang Lemma, the continuity and the coercivity of the bilinear form bring to the following error estimates from [1].

Theorem 2.14. *Let be u_h the solution of Problem 2.10. If the solution of Problem 2.3 $u \in H^{s+1}(\mathcal{K}_h)$ with $s \geq 1$ then the following error estimates hold:*

$$||u - u_h||_{ip} \leq c_1 \frac{h^{\min(p,s)}}{p^{s-1/2}} ||u||_{H^{s+1}(\mathcal{K}_h)} \ ,$$

$$||u - u_h||_{L^2(\Omega)} \leq c_0 \frac{h^{\min(p,s)+1}}{p^s} ||u||_{H^{s+1}(\mathcal{K}_h)} \ ,$$

where c_0 and c_1 do not depend on h or p.

2.4 Basis functions

In our proposal we have employed the Spectral Element Method[3] basis functions. These *orthonormal* bases are quite interesting because they are spectral and Lagragian at the same time. Lagrangian bases are easy to integrate *numerically.* On other hand, the derivatives of spectral bases are easy to obtain *analytically.* So, Spectral Element Method basis functions are easy to integrate numerically and their derivative can be recovered analytically.

These properties are not mandatory in our proposal, but they ease the development and reduce floating point operations.

Let $\mathcal{B}_N$ be a set containing the Spectral Element Method basis functions of N-degree on a reference domain

$$\mathcal{B}_N \ = \ \{ \ \psi_i : [-1,1] \to \mathbb{R} \mid \quad 0 \leq i \leq N \ \} \ ,$$

the $\psi_i : [-1,1] \to \mathbb{R}$ are defined as

$$\psi_i(z) \ = \ \frac{-1}{N(N+1)} \frac{(1-z^2)L'_N(z)}{(z-z_i)L_N(z_i)} \tag{2.11}$$

where $\{z_i\}_{0 \leq i \leq N}$ are the Gauss Lobatto Legendre nodes and L_N is the N-th degree Legendre polynomial. The derivative of ψ_i is given by

$$\frac{d\psi_i}{dz}(z) \ = \ \frac{-1}{N(N+1)L_N(z_i)(z-z_i)} \Big[(1-z^2)L''_N(z) - (2z + \tfrac{1-z^2}{z-z_i})L'_N(z) \Big] \tag{2.12}$$

A tensorial basis is a multidimensional basis whose elements are defined as tensor products of one-dimensional basis functions in each of the coordinate directions.

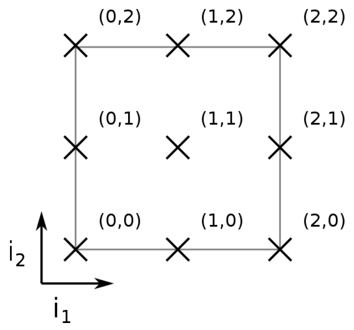


Figure 2.1: Basis functions index example.

Let $\hat{K}$ be the reference square domain $[-1,1]\times[-1,1]$. Let $\mathbf{i}=(i_1,i_2)$ be a multi-index which identifies a basis function centered in $(z_{i_1},z_{i_2})\in\hat{K}$, where z_i is the i-th Gauss Legendre Lobatto quadrature node. Figure 2.1 shows the index scheme in case of $N=2$. Let $B_N^{\hat{K}}$ be the set of the N-degree basis functions on $\hat{K}$

$$B_N^{\hat{K}} \;=\; \{\; \hat{\varphi}_{\mathbf{i}} : \hat{K}\to\mathbb{R} \mid \quad 0\le i_1\le N,\ 0\le i_2\le N \;\} \;,$$

the $\hat{\varphi}_{\mathbf{i}}:\hat{K}\to\mathbb{R}$ are the tensorial basis functions defined as follows

$$\hat{\varphi}_{\mathbf{i}}(x_1,x_2) \;=\; \psi_{i_1}(x_1)\;\psi_{i_2}(x_2)\;.$$

2.5 Symmetric Interior Penalty Galerkin method algebraic formulation

The bilinear form $a_h(\cdot,\cdot)$ of the Symmetric Interior Penalty Galerkin method for Problem 2.1 (see eq. 2.8) can be rewritten as

$$a_h(u,v)=\sum_{K\in\mathcal{K}_h}\gamma^K(u,v)+\sum_{F\in\mathcal{F}_h^i}\phi^F(u,v)+\sum_{F\in\mathcal{F}_h^b}\xi^F(u,v)\;,$$

where $\gamma^K(\cdot,\cdot)$ represents the contribution of the integral on $K\in\mathcal{K}_h$

$$\gamma^K(u,v)=\int_K \nabla u\nabla v+\lambda uv \quad ,$$

$\phi^F(\cdot,\cdot)$ represents the contribution of the integral on $F\in\mathcal{F}_h^i$

$$\phi^F(u,v)=\int_F \frac{\eta}{h_F}[\![u]\!][\![v]\!]-\{\{\nabla u\cdot\mathbf{n}_F\}\}[\![v]\!]-\{\{\nabla v\cdot\mathbf{n}_F\}\}[\![u]\!] \quad ,$$

and $\xi^F(\cdot,\cdot)$ represents the contribution of the integral on $F\in\mathcal{F}_h^b$

$$\xi^F(u,v)=\int_F \frac{\eta}{h_F}uv-\nabla u\cdot\mathbf{n}_F v-\nabla v\cdot\mathbf{n}_F u \quad .$$

Let B_N^K be the set of the N-degree basis functions on a generic $K \in \mathcal{K}_h$.

The matrix $\Gamma^K \in \mathbb{R}^{(N+1)\times(N+1)}$ associated to $\gamma^K : B_N^K \times B_N^K \to \mathbb{R}$ is defined by the entries

$$\Gamma^K_{\mathbf{ij}} \; = \; \gamma^K(\varphi_{\mathbf{i}}, \varphi_{\mathbf{j}}) \quad \forall \varphi_{\mathbf{i}},\ \varphi_{\mathbf{j}} \in B_N^K \ .$$

Let $K_1, K_2 \in \mathcal{K}_h$ such that $K_1 \cap K_2 = F \in \mathcal{F}_h^i$. The outward normal vector $\mathbf{n}_F$ is oriented form K_1 to K_2. Let B_N^F be

$$B_N^F \; = \; B_N^{K_1} \; \cup \; B_N^{K_2} \ .$$

The matrix $\Phi^F \in \mathbb{R}^{2(N+1)\times 2(N+1)}$ associated to $\phi^F : \ B_N^F \times B_N^F \to \mathbb{R}$ is defined by the entries

$$\Phi^F_{\mathbf{ij}} = \phi^F(\varphi_{\mathbf{i}}, \varphi_{\mathbf{j}}) \quad \forall \varphi_{\mathbf{i}},\ \varphi_{\mathbf{j}} \in B_N^F \ ,$$

Φ^F can be partitioned in

$$\Phi^F_{\mathbf{ij}} \; = \; \begin{cases} \Phi^F_{\mathbf{ij}}|_+^+ = \phi^F(\varphi_{\mathbf{i}}, \varphi_{\mathbf{j}}) & \text{if } \varphi_{\mathbf{i}},\ \varphi_{\mathbf{j}} \in B_N^{K_1} \\ \\ \Phi^F_{\mathbf{ij}}|_-^+ = \phi^F(\varphi_{\mathbf{i}}, \varphi_{\mathbf{j}}) & \text{if } \varphi_{\mathbf{i}} \in B_N^{K_1},\ \varphi_{\mathbf{j}} \in B_N^{K_2} \\ \\ \Phi^F_{\mathbf{ij}}|_-^- = \phi^F(\varphi_{\mathbf{i}}, \varphi_{\mathbf{j}}) & \text{if } \varphi_{\mathbf{i}},\ \varphi_{\mathbf{j}} \in B_N^{K_2} \\ \\ \Phi^F_{\mathbf{ij}}|_+^- = \phi^F(\varphi_{\mathbf{i}}, \varphi_{\mathbf{j}}) & \text{if } \varphi_{\mathbf{i}} \in B_N^{K_2},\ \varphi_{\mathbf{j}} \in B_N^{K_1} \quad . \end{cases}$$

The matrices $\Phi^F|_+^+,\ \Phi^F|_-^+,\ \Phi^F|_-^-,\ \Phi^F|_+^- \in \ \mathbb{R}^{(N+1)\times(N+1)}$ are defined as

$$\Phi^F_{\mathbf{ij}}|_+^+ \;=\; \int_F \frac{\eta}{h_F}\,\varphi_{\mathbf{i}}\,\varphi_{\mathbf{j}} \;-\frac{1}{2}\,\nabla\varphi_{\mathbf{i}}\cdot\mathbf{n}_F\,\varphi_{\mathbf{j}} \;-\frac{1}{2}\,\nabla\varphi_{\mathbf{j}}\cdot\mathbf{n}_F\,\varphi_{\mathbf{i}}\,dx \qquad \forall\varphi_{\mathbf{i}},\ \varphi_{\mathbf{j}} \in B_N^{K_1} \ ,$$

$$\Phi^F_{\mathbf{ij}}|_-^- \;=\; \int_F \frac{\eta}{h_F}\,\varphi_{\mathbf{i}}\,\varphi_{\mathbf{j}} \;+\frac{1}{2}\,\nabla\varphi_{\mathbf{i}}\cdot\mathbf{n}_F\,\varphi_{\mathbf{j}} \;+\frac{1}{2}\,\nabla\varphi_{\mathbf{j}}\cdot\mathbf{n}_F\,\varphi_{\mathbf{i}}\,dx \qquad \forall\varphi_{\mathbf{i}},\ \varphi_{\mathbf{j}} \in B_N^{K_2} \ ,$$

$$\Phi^F_{\mathbf{ij}}|_-^+ \;=\; \int_F -\frac{\eta}{h_F}\,\varphi_{\mathbf{i}}\,\varphi_{\mathbf{j}} \;+\frac{1}{2}\,\nabla\varphi_{\mathbf{i}}\cdot\mathbf{n}_F\,\varphi_{\mathbf{j}} \;-\frac{1}{2}\,\nabla\varphi_{\mathbf{j}}\cdot\mathbf{n}_F\,\varphi_{\mathbf{i}}\,dx \qquad \forall\varphi_{\mathbf{i}} \in B_N^{K_1},\ \forall\varphi_{\mathbf{j}} \in B_N^{K_2} \ ,$$

$$\Phi^F_{\mathbf{ij}}|_+^- \;=\; \int_F -\frac{\eta}{h_F}\,\varphi_{\mathbf{i}}\,\varphi_{\mathbf{j}} \;-\frac{1}{2}\,\nabla\varphi_{\mathbf{i}}\cdot\mathbf{n}_F\,\varphi_{\mathbf{j}} \;+\frac{1}{2}\,\nabla\varphi_{\mathbf{j}}\cdot\mathbf{n}_F\,\varphi_{\mathbf{i}}\,dx \qquad \forall\varphi_{\mathbf{i}} \in B_N^{K_2},\ \forall\varphi_{\mathbf{j}} \in B_N^{K_1} \ .$$

Let $K \cap \partial\Omega = F \in \mathcal{F}_h^b$. The matrix $\Xi^F \in \mathbb{R}^{(N+1)\times(N+1)}$ associated to $\xi^F : B_N^K \times B_N^K \to \mathbb{R}$ is defined by the entries

$$\Xi^F_{\mathbf{ij}} = \xi(\varphi_{\mathbf{i}}, \varphi_{\mathbf{j}}) \quad \forall \varphi_{\mathbf{i}},\ \varphi_{\mathbf{j}} \in B_N^K \ .$$

2.5.1 Numerical quadrature of matrices

In this section, we show the numerical quadrature of Γ^K. The numerical quadratures of Φ^F and Ξ^F follow a similar procedure.

We want express Γ^K in function of the reference domain $\hat{K}$. So, let $\mathbf{F}^K : \hat{K} \to K$ be an invertible linear map which transforms $\hat{K}$ in $K \in \mathcal{K}_h$. The Jacobian matrix of $\mathbf{F}^K$ is defined by the entries

$$J^K_{ij} = \frac{\partial F^K_i}{\partial \hat{x}_j} \ ,$$

the inverse matrix of J^K is called J^{-K}.

If we define S_1, S_2 and S_3 as

$$\begin{aligned} S_1 &= \left[\, {J^{-K}_{11}}^2 \; + \; {J^{-K}_{12}}^2 \, \right] |\det J^K| \ , \\ S_2 &= \left[\, {J^{-K}_{21}}^2 \; + \; {J^{-K}_{22}}^2 \, \right] |\det J^K| \ , \\ S_3 &= |\det J^K| \ , \end{aligned}$$

the $\Gamma^K_{\mathbf{ij}}$ becomes

$$\Gamma^K_{\mathbf{ij}} \; = \; \int_{\hat{K}} S_1 \, \frac{\partial \hat{\varphi}_{\mathbf{i}}}{\partial \hat{x}_1}(\hat{\mathbf{x}}) \frac{\partial \hat{\varphi}_{\mathbf{j}}}{\partial \hat{x}_1}(\hat{\mathbf{x}}) + S_2 \, \frac{\partial \hat{\varphi}_{\mathbf{i}}}{\partial \hat{x}_2}(\hat{\mathbf{x}}) \frac{\partial \hat{\varphi}_{\mathbf{j}}}{\partial \hat{x}_2}(\hat{\mathbf{x}}) + S_3 \, \hat{\lambda}(\hat{\mathbf{x}}) \, \hat{\varphi}_{\mathbf{i}}(\hat{\mathbf{x}}) \, \hat{\varphi}_{\mathbf{j}}(\hat{\mathbf{x}}) \; d\hat{\mathbf{x}} \qquad \forall \hat{\varphi}_{\mathbf{i}}, \, \hat{\varphi}_{\mathbf{j}} \in B^{\hat{K}}_N$$

Furthermore, we expand the tensorial basis functions obtaining

$$\begin{aligned} \Gamma^K_{\mathbf{ij}} \; = \; \int_{\hat{K}} \Big\{ \quad & S_1 \, \frac{\partial \psi_{i_1}}{\partial \hat{x}_1}(\hat{x}_1) \frac{\partial \psi_{j_1}}{\partial \hat{x}_1}(\hat{x}_1) \psi_{i_2}(\hat{x}_2) \psi_{j_2}(\hat{x}_2) \\ & + S_2 \, \frac{\partial \psi_{i_2}}{\partial \hat{x}_2}(\hat{x}_2) \frac{\partial \psi_{j_2}}{\partial \hat{x}_2}(\hat{x}_2) \psi_{i_1}(\hat{x}_1) \psi_{j_1}(\hat{x}_1) \\ & + S_3 \, \hat{\lambda}(\hat{\mathbf{x}}) \; \psi_{i_1}(\hat{x}_1) \; \psi_{j_1}(\hat{x}_1) \; \psi_{i_2}(\hat{x}_2) \; \psi_{j_2}(\hat{x}_2) \quad \Big\} \; d\hat{\mathbf{x}} \end{aligned}$$

Finally, we integrate numerically $\Gamma^K_{\mathbf{ij}}$ using the Gauss Legendre Lobatto quadrature rule with $(N+1)$ nodes.

$$\Gamma^K_{\mathbf{ij}} \quad = \quad \begin{cases} w_{i_2} \; S_1 \displaystyle\sum_{p=0}^{N} w_p \Big[\frac{\partial \psi_{i_1}}{\partial \hat{x}_1}(z_p) \Big]^2 & \\ \quad + \; w_{i_1} \; S_2 \displaystyle\sum_{q=0}^{N} w_q \Big[\frac{\partial \psi_{i_2}}{\partial \hat{x}_2}(z_q) \Big]^2 & \text{if } i_1 = j_1 \text{ and } i_2 = j_2 \\ \quad + \; w_{i_1} \; w_{i_2} \; S_3 \; \hat{\lambda}(z_{i_1}, z_{i_2}) & \\ & \\ w_{i_2} \; S_1 \displaystyle\sum_{p=0}^{N} w_p \frac{\partial \psi_{i_1}}{\partial \hat{x}_1}(z_p) \frac{\partial \psi_{j_1}}{\partial \hat{x}_1}(z_p) & \text{if } i_2 = j_2 \text{ and } i_1 \neq j_1 \\ & \\ w_{i_1} \; S_2 \displaystyle\sum_{q=0}^{N} w_q \frac{\partial \psi_{i_2}}{\partial \hat{x}_2}(z_q) \frac{\partial \psi_{j_2}}{\partial \hat{x}_2}(z_q) & \text{if } i_1 = j_1 \text{ and } i_2 \neq j_2 \\ & \\ 0 & \text{otherwise} \end{cases}$$

where z_i and w_i are the i-th node and weight of the Gauss Legendre Lobatto quadrature rule. If $\lambda \neq 0$, the integrand degree is higher than $2N-1$, then the quadrature is not exact. Figure 2.2 shows the sparsity of Γ^K.

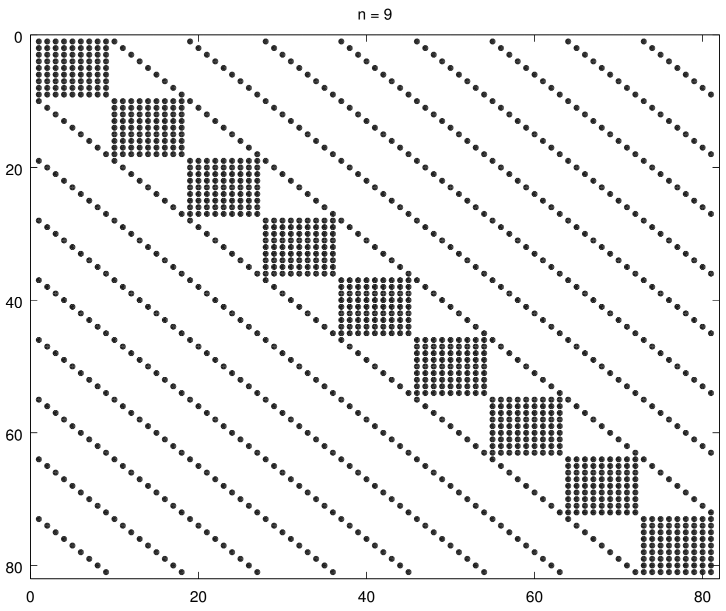


Figure 2.2: Sparsity of Γ^K.

Chapter 3

A proposal for the implementation of Discontinuous Galerkin methods on GPUs

The numerical solution of a Finite Element Method problem consists of three main steps[3]:

1. Mesh generation. The domain of the problem is approximated by a mesh.
2. The creation of the stiffness matrix and of the right-hand side vector. Which is split in:
   (a) The generation of local components. The differential operators are discretized on each element of the mesh and the local matrices and the local right-hand sides are computed.
   (b) The assembly of the global matrix. The summation of the local components produces the stiffness matrix and global right-hand side.
3. The solution of the linear system. An iterative solver usually provides the solution of the algebraic system.

The implementation on GPUs of the above steps encounter two critical points:

- the assembly of the global stiffness matrix, which implies non-homogeneous memory accesses to data locations assigned to the degrees of freedom shared by different elements;
- the storage and the algebraic operations (such as the matrix-vector product) of a sparse stiffness matrix.

Both points make difficult to fully exploit the memory bandwidth of GPUs because memory operations do not coalesce.

Our proposal consists in merging the generation of the local components and the assembly of the global matrix into the matrix-vector product. In such a way, we avoid the storage of the global stiffness matrix, saving device memory and preventing non-coalescing memory accesses.

We employ a Discontinuous Galerkin method because, in our opinion, these methods have an attractive property: every mesh element interacts with the neighbor elements only through boundary integrals. In this way, each mesh element has its nodes. As opposite, different elements share the same nodes in Continuous Galerkin methods.

The simplest Discontinuous Galerkin method for elliptic problems is the Symmetric Interior Penalty Galerkin method.

Since we want to recompute the local components at every matrix-vector product, we decide to employ spectral tensorial basis functions in order to take advantage of their computational simplicity in case of quadrilateral meshes. In particular, we have chosen the Spectral Element Method basis functions because they are Lagrangian.

In order to solve the linear system we have implemented an iterative method: the Conjugate Gradient. Our goal is to employ the GPU in order to perform the matrix-vector product and the other vector operations, necessary to solve the linear system.

3.1 Data structure

Let $\mathcal{K}_h$ be a structured quadrilateral mesh of size $[row \times col]$, where row are the number of rows and col are the number of columns. For the two-dimensional problem, solved with N-degree basis functions, we save the vector of the degrees of freedom using a three-dimensional data structure on the device memory. This data structure has three strides: the first is 1, the second is the number of elements per row (col), the third is the total number of mesh elements ($row \cdot col$).

So, we have an array which is mapped by three indexes, $(x, y, \mathbf{i})$. The first two identify the coordinates of the mesh element; the third index identifies a specific degree of freedom inside the element. The global index of the array is computed as

$$(x, y, \mathbf{i}) = x + (\ col \cdot y\ ) + \{\ row \cdot col \cdot [\ (N+1)\ i_1 + i_2\ ]\ \}$$

This data structure is fundamental to avoid thread divergence and minimize non-coalescing memory accesses. If every thread computes the information related to a mesh element, it does the same operation at the same time, so there is not divergence. Furthermore, every thread reads/writes the data of the same degree of freedom within an element at the same time.

3.2 The matrix vector product

The way we perform the stiffness matrix-vector product plays the central role in the implementation of our proposal. Our code does not store the global stiffness matrix in memory. The values of the local matrices Γ^K, Φ^F and Ξ^F (see Section 2.5) are recomputed on the fly every time a matrix-vector product is needed. Let A be the global stiffness matrix, the matrix-vector product,

$$\vec{y} \ = \ A\,\vec{x}\ ,$$

is performed applying the actions of the local matrices to the multiplied vector $\vec{x}$. The result is stored in the solution vector $\vec{y}$. In this way, the matrix-vector product involves the storage of only two vectors and of the mesh informations. Let us define

$$\mathcal{I}^K = \{\ \mathbf{i}\ |\quad 0 \le i_1 \le N,\ 0 \le i_2 \le N\ \}$$

as the set which contains the multi-indexes associated to the degrees of freedom of $K \in \mathcal{K}_h$.

Let $x(K, \mathbf{i})$ be the $\vec{x}$ vector value associated to the degree of freedom $\mathbf{i} \in \mathcal{I}^K$ into $K \in \mathcal{K}_h$. In order to compute $y(K_0, \mathbf{i})$, the kernels perform this computation

$$\begin{aligned}
y(K_0,\mathbf{i}) \;=\; & \sum_{\mathbf{j}\in\mathcal{I}^{K_0}} \left(\Gamma^{K0}_{\mathbf{ij}} + \Phi^{F_1}_{\mathbf{ij}}|^{-}_{-} + \Phi^{F_2}_{\mathbf{ij}}|^{+}_{+} + \Phi^{F_3}_{\mathbf{ij}}|^{+}_{+} + \Phi^{F_4}_{\mathbf{ij}}|^{-}_{-} \right) x(K_0,\mathbf{j}) \\
& + \sum_{\mathbf{j}\in\mathcal{I}^{K_1}} \Phi^{F_1}_{\mathbf{ij}}|^{-}_{+}\, x(K_1,\mathbf{j}) \\
& + \sum_{\mathbf{j}\in\mathcal{I}^{K_2}} \Phi^{F_2}_{\mathbf{ij}}|^{+}_{-}\, x(K_2,\mathbf{j}) \\
& + \sum_{\mathbf{j}\in\mathcal{I}^{K_3}} \Phi^{F_3}_{\mathbf{ij}}|^{+}_{-}\, x(K_3,\mathbf{j}) \\
& + \sum_{\mathbf{j}\in\mathcal{I}^{K_4}} \Phi^{F_4}_{\mathbf{ij}}|^{-}_{+}\, x(K_4,\mathbf{j})
\end{aligned}$$

where $K_1,\ K_2,\ K_3,\ K_4 \in \mathcal{K}_h$ share respectively $F_1,\ F_2,\ F_3,\ F_4 \in \mathcal{F}^i_h$ with K_0. The normal vectors $\mathbf{n}_{F_1}$ and $\mathbf{n}_{F_4}$ point inside K_0. The normal vectors $\mathbf{n}_{F_2}$ and $\mathbf{n}_{F_3}$ point outside K_0.

If K shares a face with $\partial\Omega$, for instance $K \cap \partial\Omega = F_1 \in \mathcal{F}^b_h$, the value of $y(K_0, \mathbf{i})$ is computed as

$$\begin{aligned}
y(K_0,\mathbf{i}) \;=\; & \sum_{\mathbf{j}\in\mathcal{I}^{K_0}} \left(\Gamma^{K0}_{\mathbf{ij}} + \Xi^{F_1}_{\mathbf{ij}} + \Phi^{F_2}_{\mathbf{ij}}|^{+}_{+} + \Phi^{F_3}_{\mathbf{ij}}|^{+}_{+} + \Phi^{F_4}_{\mathbf{ij}}|^{-}_{-} \right) x(K_0,\mathbf{j}) \\
& + \sum_{\mathbf{j}\in\mathcal{I}^{K_2}} \Phi^{F_2}_{\mathbf{ij}}|^{+}_{-}\, x(K_2,\mathbf{j}) \\
& + \sum_{\mathbf{j}\in\mathcal{I}^{K_3}} \Phi^{F_3}_{\mathbf{ij}}|^{+}_{-}\, x(K_3,\mathbf{j}) \\
& + \sum_{\mathbf{j}\in\mathcal{I}^{K_4}} \Phi^{F_4}_{\mathbf{ij}}|^{-}_{+}\, x(K_4,\mathbf{j})
\end{aligned}$$

3.3 Code

Before a brief description of the code, it is important to remember that there are two memory contexts, one for the host and one for the device. For example, we cannot use pointer and variables of the host on the device and vice-versa. So, every host class, which stores data used by the device, has a paired device class. This is the same design concept employed by the Thrust library[24], where the `host_vector` class is paired to the `device_vector` class.

For example, in our code the `host_mode_vector` class stores the vector of the degrees of freedom on the host, while the `mode_vector` class is its device counterpart.

On other hand the case of `square_mesh` class is slightly different. This class computes the data related to a square mesh. This data is passed to `quadrilateral_mesh_info`, which copies the needed data on the device. In case we want to design a more generic quadrilateral mesh class, we will pass anyway the data to `quadrilateral_mesh_info`. This class is generic enough to deal with any kind of structured quadrilateral mesh.

The class sipg_sem_2d defines a Poisson problem with Dirichlet boundary conditions solved with the Symmetric Interior Penalty Galerkin method on a unit square domain. This class computes the right hand side of vector, initializes data on the device, provides the implementation of matrix-vector product and calls the appropriate iterative method.

The class sipg_sem_2d_multigpu does the same tasks of sipg_sem_2d on multiple GPUs interconnected with MPI. In order to exchange the halos of the global vector of the degrees of freedom between multiple GPUs, we employ the gcl library[6].

Matrix-vector product function applies the following kernels to the mode_vector of the degrees of freedom: volume kernel adds the contributions related to the integral on the mesh element areas, flux6a and flux6b kernels add the contributions related to the integral on mesh element borders.

The abstract class abs_mvm provides the interface of the matrix-vector product in order to generalize the implementation of the iterative solvers.

The function conjugate_gradient solves the linear system. It is called by the sipg_sem_2d constructor.

We have seen that the computation of the local Laplacian operator requires a large amount of computation time when the basis degree is high. So, we design the mode_matrix class which is able to store local operators on the device in order to avoid re-computing them every time. The mode_matrix class host counterpart is the host_mode_matrix class.

The code has been documented with Doxygen. For furthers details, it is possible to refer to the generated documentation.

3.4 GPU kernels

In this section we describe briefly the kernels developed. We consider five kernels.

Three of them, volume_mvm< ,1>, volume_mvm< ,0> and volume do the same operation in different ways. They apply the local matrices Γ^K to the input mode_vector and write the solution to the output mode_vector.
They are:

- volume_mvm< ,1> applies the same local Laplacian operator to all the mesh elements, this local operator is saved previously on a mode_matrix object.This kind of kernel can be employed only when all the mesh elements have the same geometry.
- volume_mvm< ,0> applies the local Laplacian operators to the mesh elements, every local operator is saved previously in a mode_matrix object. This object can require a large amount of memory. So, we are not able to run tests with a large number of mesh elements or an high degree of basis functions.
- volume applies the local Laplacian operators to the mesh elements. Each local operator is recomputed every time as needed.

The flux6a and flux6b kernels, run one after the other, recompute and apply Φ^F and Ξ^F to the input mode_vector. The solution is written to the output mode_vector.

Chapter 4

Validation of the code

In this chapter we will show the validation tests introducing you the results of the test made. First of all, we have tested the correctness of Gauss Legendre Lobatto quadrature nodes and weights. Then, we have proved the correctness of the local Laplacian operator solving the Poisson problem using the Galerkin Spectral method. Finally, we have validated the Symmetric Interior Penalty Galekin method solving the Poisson problem with an increasing number of mesh elements.

4.1 Test of the numerical quadrature

The Gauss Legendre Lobatto nodes $\{x_i\}_{0\leq i\leq N}$ are the roots of the derivative of L_N. For $N \leq 6$, we have validated the node values confronting them with their analytical counterpart. The errors encountered are negligible, they have the same magnitude of the machine epsilon (approximately 10^{-16}).

Furthermore, we have check the quadrature error of x^p varying the degree p and the number of quadrature node n. The Gauss Legendre Lobatto quadrature rule performs an exact integration for polynomials up to the degree $2n-3$. The quadrature errors have been calculated as:

$$q_{err} \;=\; \Big|\; \int_a^b x^p dx \;-\; \sum_i^n \frac{b-a}{2}\; w_i \;\Big[\; \frac{b-a}{2} x_i + \frac{a+b}{2}\; \Big]^p \;\Big|$$

Table 4.1 shows the quadrature errors. The cells with gray background contain the errors related to cases in which the quadrature is exact, i.e. $p \leq 2n-3$. The values in these cells are zeros or close to the machine epsilon.

p	n = 2	n = 3	n = 4	n = 5	n = 6	n = 7	n = 8
1	0	0	2.22045e-16	0	0	-2.22045e-16	-2.22045e-16
2	0.166667	0	0	-4.44089e-16	0	-4.44089e-16	-4.44089e-16
3	0.75	0	4.44089e-16	-4.44089e-16	0	-8.88178e-16	-4.44089e-16
4	2.3	0.00833333	0	-8.88178e-16	-8.88178e-16	-8.88178e-16	-1.77636e-15
5	6	0.0625	0	0	0	0	-1.77636e-15
6	14.3571	0.284226	0.00047619	0	3.55271e-15	-3.55271e-15	0
7	32.625	1.01562	0.005	0	3.55271e-15	-3.55271e-15	0
8	71.7222	3.14149	0.0302222	2.83447e-05	7.10543e-15	-1.42109e-14	-7.10543e-15
9	154.2	8.82891	0.138	0.000382653	1.42109e-14	-1.42109e-14	-1.42109e-14
10	326.409	23.1858	0.528824	0.00288646	1.71786e-06	-2.84217e-14	-2.84217e-14
11	683.25	57.915	1.7956	0.0160578	2.83447e-05	-5.68434e-14	0
12	1418.42	139.254	5.5798	0.0734963	0.000256313	1.05114e-07	0
13	2926.29	325.032	16.2056	0.293132	0.00168178	2.04972e-06	0
14	6008.03	740.986	44.6314	1.05483	0.00895392	2.1609e-05	6.46878e-09

Table 4.1: Quadrature errors for $a = 1$ and $b = 2$.

4.2 Validation of SEM basis functions

The implementation of the *SEM* basis function values (see Eq. 2.11) are easy to validate. This kind of basis is Lagrangian, i.e. the value of a given basis function equals to 1 at its corresponding Gauss Legendre Lobatto node and 0 in the others.

However, in order to test the correctness of the basis function derivatives (see Eq. 2.12), we have employed them in the solution of a Poisson problem with Dirichlet boundary conditions and then we have checked if the convergence rate of error corresponds to the theory.

Problem 4.1. *Find $u : \Omega \to \mathbb{R}$ such that*

$$\begin{aligned} -\Delta u &= f \ \textit{in } \Omega \\ u &= g \ \textit{in } \partial\Omega \end{aligned}$$

where $g : \partial\Omega \to \mathbb{R}$ is the Dirichlet boundary condition function.

We discretized Problem 4.1 with the Galerkin Spectral Method, on a unit square domain $\Omega = (0, 1)^2$, with boundary conditions weakly imposed by the Nitsche method [10]. Let φ_i be the *SEM* basis functions (2.11), the solution set V_N is defined as

$$V_N(\Omega) = \left\{ \quad v(\mathbf{x}) = \sum_{i_1,i_2=0}^{N} a_{i_1 i_2} \varphi_{i_1}(x_1)\varphi_{i_2}(x_2) \quad \middle| \quad a_{i_1 i_2} \in \mathbb{R} \quad \right\}$$

So, we have the following discretization of Problem 4.1.

Problem 4.2. *Find $u_N \in V_N$ such that*

$$\int_\Omega \nabla u_N \nabla v_N - \int_{\partial\Omega} \frac{\partial u_N}{\partial n} v_N - \int_{\partial\Omega} \frac{\partial v_N}{\partial n} u_N + \int_{\partial\Omega} \mu u_N v_N = \int_\Omega f v_N - \int_{\partial\Omega} \frac{\partial v_N}{\partial n} g + \int_{\partial\Omega} \mu g v_N \quad \forall v \in V_N$$

where $\mu = 100$ is the penalization term.

Notice that in the Symmetric Interior Penalty Galerkin method, the Dirichlet boundary conditions are imposed with the Nitsche method. So, Problem 4.2 is equivalent to the *SIPG* on a single-element mesh.

	u_{ex}
A	$-2x(y-1)(y-2x+xy+2)e^{x-y}$
B	$\sin(2\pi y)\left[(4-3x-9x^2)-4\pi^2(x-x^2)\right]e^{3x}$
C	$2x^3-6xy(1-y)$
D	$-2y^2-2x^2$
E	$64\pi^2\left[\,\cos(8\pi x)+\cos(8\pi y)\,\right]$

Table 4.2: Analytical solutions chosen for testing.

Right-hand side functions f are chosen in order to produce solutions shown in Table 4.2. In Spectral Methods with analytical solutions, if we increase the degree of basis functions the order of convergence of the error is exponential [3], i.e.

$$\exists \lambda > 0 : \quad ||u - u_N||_{H^1(\Omega)} \leq Ce^{-\lambda N} \ . \tag{4.1}$$

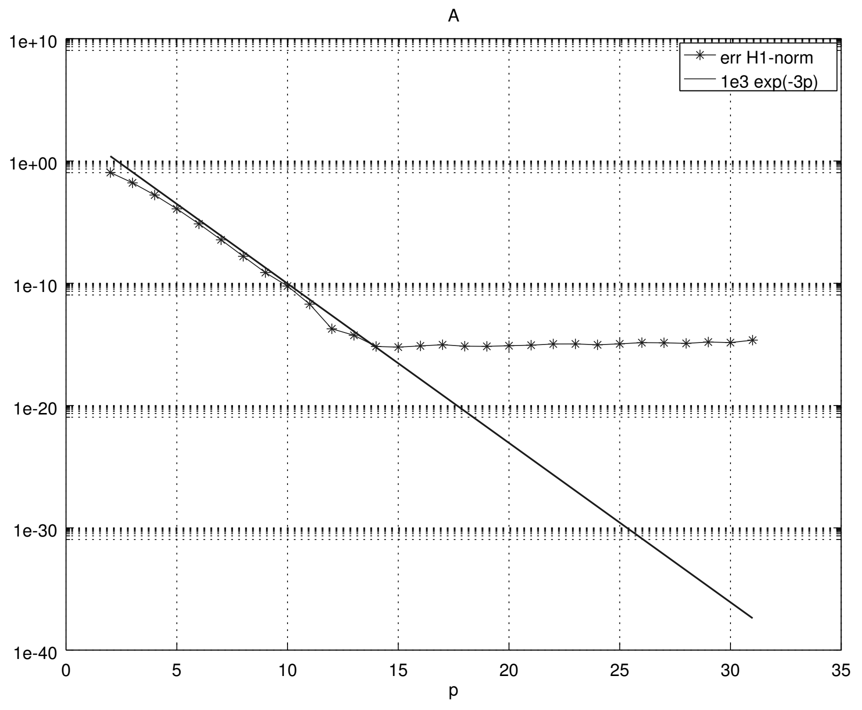
Table 4.3 and Table 4.4 show the errors in L^∞, L^2 and H^1 norms for solutions A, B and E. In order to check if the convergence rates satisfy (4.1) we plot the H^1-norm of the error varying the basis degree (see Figure 4.1 and Figure 4.2). The errors decay is given by exponential functions, confirming the correctness of the solution procedure.

Table 4.5 shows the errors for solutions C and D, which are polynomial of degree 2 and 3. In such case Problem 4.2 provides the exact solution if the basis degree p is equal or above the polynomial degree of the solution.

The linear systems resulting from these problems have been solved using the Conjugate Gradient method. The minimum error reached depends on the `toll` value utilized in Equation A.3. In these test, `toll=1e-15`.

p	$\|\|u-u_{ex}\|\|_{H^1}$	$\|\|u-u_{ex}\|\|_{L^2}$	$\|\|u-u_{ex}\|\|_{L^\infty}$	$\|\|u-u_{ex}\|\|_{H^1}$	$\|\|u-u_{ex}\|\|_{L^2}$	$\|\|u-u_{ex}\|\|_{L^\infty}$
2	0.114391	0.00552991	0.0081507	5.74799	1.41505e-13	2.0955e-13
3	0.0163707	0.000269219	0.000528206	4.66357	0.216429	0.290032
4	0.00164445	1.74668e-05	4.12153e-05	2.11731	0.0321763	0.0638047
5	0.000122799	1.2035e-06	3.83018e-06	0.419806	0.0129717	0.0241231
6	7.25018e-06	7.51039e-08	2.57521e-07	0.190424	0.00141743	0.00910705
7	3.5924e-07	4.19763e-09	1.66609e-08	0.0242083	0.000582868	0.00177117
8	1.60446e-08	2.19451e-10	9.19489e-10	0.0102321	0.00012613	0.000794347
9	7.52136e-10	1.20523e-11	5.67913e-11	0.00226715	5.16219e-05	0.000237455
10	5.93787e-11	9.81761e-13	4.46181e-12	0.00100279	1.64242e-05	0.0001162
11	1.9619e-12	2.81412e-14	1.32027e-13	1.53893e-05	2.94452e-07	1.73046e-06
12	1.89291e-14	2.24309e-16	1.0363e-15	9.78125e-06	1.07707e-07	8.32231e-07
13	5.57246e-15	1.31835e-16	2.7843e-15	1.12671e-07	1.39911e-09	1.02568e-08
14	6.81331e-16	1.09723e-16	3.05311e-16	8.92785e-08	5.8355e-10	4.81771e-09
15	6.03145e-16	8.03587e-17	1.94289e-16	1.02395e-09	7.95758e-12	6.65936e-11
16	7.577e-16	1.14589e-16	2.498e-16	8.33071e-10	3.58572e-12	3.20909e-11
17	9.42908e-16	1.21889e-16	3.26128e-16	8.05914e-12	4.20763e-14	3.91003e-13
18	7.1619e-16	8.60582e-17	2.77556e-16	6.48645e-12	2.00699e-14	1.88481e-13
19	6.95441e-16	7.3981e-17	2.08167e-16	5.56562e-14	1.26016e-15	4.44089e-15
20	7.82824e-16	6.55941e-17	2.22045e-16	5.18227e-14	1.38808e-15	4.66294e-15
21	8.67339e-16	1.12023e-16	3.60822e-16	1.74146e-14	9.77353e-16	3.10862e-15

Table 4.3: Error norms of the Spectral Galerkin method (with *SEM* basis) for solutions A (left) and B (right).



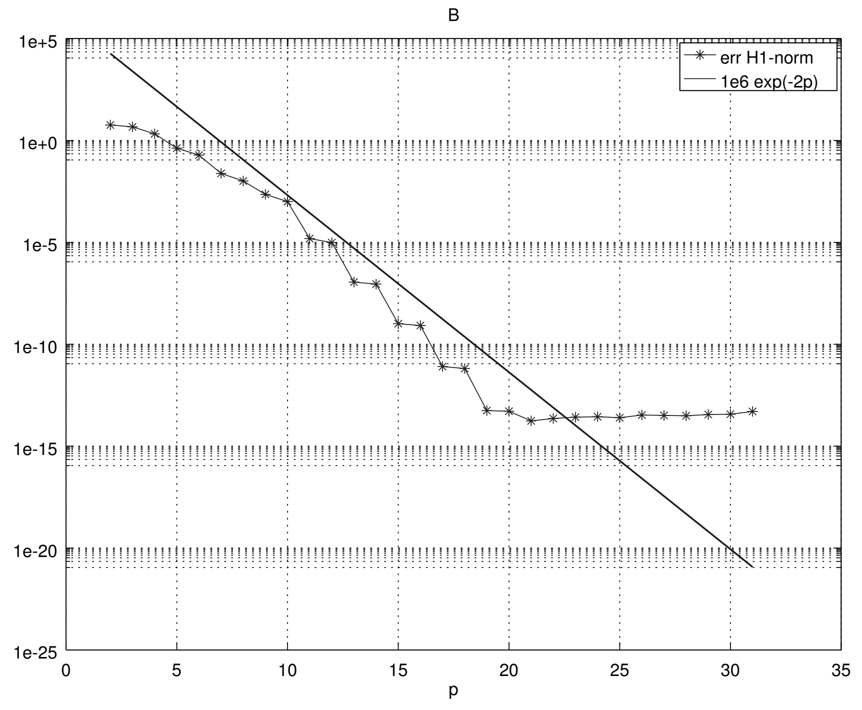


Figure 4.1: Plots of the error H^1-norm for solutions A (up) and B (down). Data from Table 4.3. Errors are bound by equation 4.1: $C_A = 10^3$, $\lambda_A = 3$, $C_B = 10^6$ and $\lambda_B = 2$

p	$\|\|u - u_{ex}\|\|_{H^1}$	$\|\|u - u_{ex}\|\|_{L^2}$	$\|\|u - u_{ex}\|\|_{L^\infty}$
2	212.69	52.2787	78.4085
3	200.826	41.5609	49.8717
4	83.9379	21.573	48.3239
5	220.874	46.3799	71.9997
6	200.288	40.986	77.9878
7	116.276	20.1787	39.2002
8	69.4052	6.98724	20.4856
9	59.4657	2.30081	5.67789
10	662.994	11.8955	56.9258
11	57.9921	0.717825	3.97058
12	24.951	0.310093	5.8162
13	182.384	1.52493	77.8771
14	5.65199	0.0458003	0.830554
15	4.08454	0.0191475	0.22007
16	1.00745	0.00556498	0.0666677
17	0.773952	0.00258097	0.0239675
18	0.143806	0.000572247	0.00619565
19	0.110976	0.000278689	0.00247738
20	0.0162029	4.87506e-05	0.000516946
21	0.0124561	2.44973e-05	0.000218686
22	0.0014731	3.47696e-06	3.71841e-05
23	0.00112619	1.78601e-06	1.62925e-05
24	0.000110372	2.10261e-07	2.2958e-06
25	8.39104e-05	1.09776e-07	1.0303e-06
26	6.93848e-06	1.09105e-08	1.22275e-07
27	5.2479e-06	5.76797e-09	5.58323e-08
28	3.71503e-07	4.91024e-10	5.66021e-09
29	2.79694e-07	2.62163e-10	2.61844e-09
30	1.71572e-08	1.93472e-11	2.29521e-10
31	1.28646e-08	1.04145e-11	1.07283e-10
32	6.90852e-10	6.72649e-13	8.21632e-12
33	5.16e-10	3.63707e-13	3.86291e-12
34	2.44984e-11	2.44395e-14	2.58016e-13
35	1.82728e-11	1.58245e-14	1.31894e-13
36	7.82097e-13	7.62064e-15	1.93179e-14
37	4.77952e-13	1.04129e-14	3.61933e-14
38	1.69497e-13	5.1081e-15	2.02061e-14
39	2.00936e-13	8.4806e-15	2.63123e-14

Table 4.4: Error norms of the Spectral Galerkin method (with *SEM* basis) for solution (E).

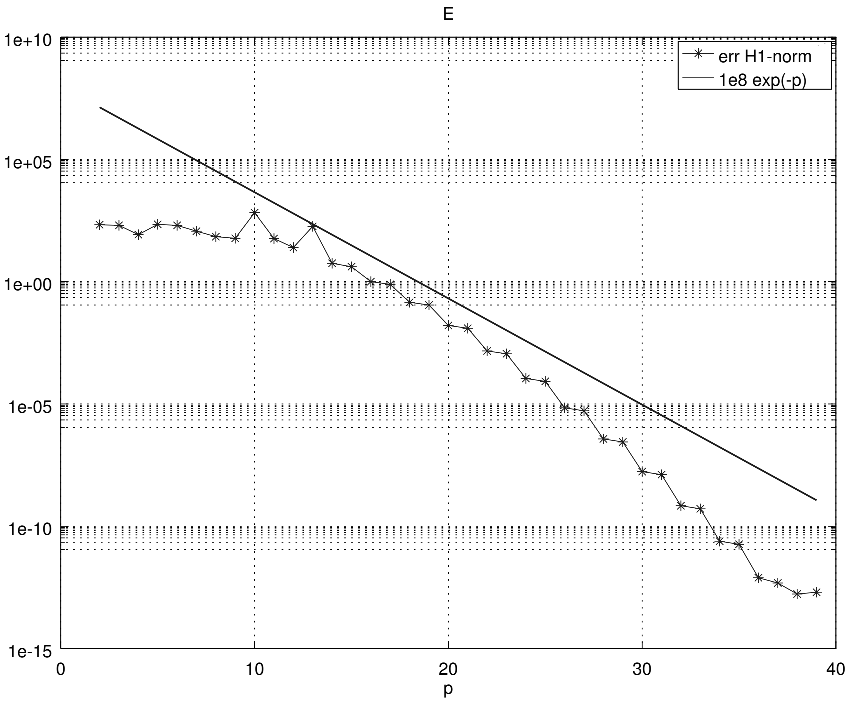


Figure 4.2: Plots of the error H^1-norm for solution E. Data from Table 4.4 Errors are bound by equation 4.1: $C = 10^8$ and $\lambda = 1$.

p	$\|\|u - u_{ex}\|\|_{H^1}$	$\|\|u - u_{ex}\|\|_{L^2}$	$\|\|u - u_{ex}\|\|_{L^\infty}$	$\|\|u - u_{ex}\|\|_{H^1}$	$\|\|u - u_{ex}\|\|_{L^2}$	$\|\|u - u_{ex}\|\|_{L^\infty}$
2	0.0722202	0.000593245	0.00125845	1.03965e-16	1.90809e-17	5.55112e-17
3	1.45521e-16	1.43761e-17	5.55112e-17	4.20877e-16	4.78204e-17	1.11022e-16
4	2.28119e-16	4.44997e-17	1.11022e-16	2.30566e-15	1.8879e-16	9.99201e-16
5	4.64418e-16	4.87124e-17	1.11022e-16	2.01928e-15	1.86356e-16	4.44089e-16
6	5.38793e-16	5.31445e-17	1.11022e-16	3.26023e-15	3.09804e-16	8.88178e-16

Table 4.5: Error norms of the Spectral Galerkin method (with *SEM* basis) for solutions C (left) and D (right).

4.3 Validation of the Symmetric Interior Penalty Galerkin method

In this section we show the numerical result of Problem 4.1 on $\Omega = (0,1)^2$ solved with the Symmetric Interior Penalty Galerking method.

The Ω domain is discretized by a regular conforming mesh made of square elements (see Figure 4.3). The f functions are chosen in order to obtain the solutions shown in table 4.2. The penalization term η equals to $100p^2$, where p is the basis degree.

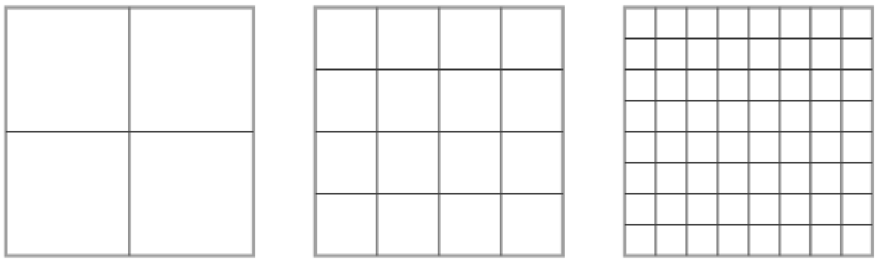


Figure 4.3: Example of meshes employed in the SIPG validation tests. Left $h = 1/2$, center $h = 1/4$ and right $h = 1/8$.

In our tests, we consider the empirical convergence rates in both the energy norm and the L^2-norm of the errors. The $|||\cdot|||_{ip}$-norm the $||\cdot||_{H^1}$-norm are equal in case of continuous solutions. Therefore, in these tests the H^1-norm is the energy norm. We compute the empirical convergence rates as

$$\beta_0 = \log_2\left(\frac{||u_h - u_{ex}||_{L^2(\Omega)}}{||u_{h/2} - u_{ex}||_{L^2(\Omega)}} \right),$$
$$\beta_1 = \log_2\left(\frac{||u_h - u_{ex}||_{H^1(\Omega)}}{||u_{h/2} - u_{ex}||_{H^1(\Omega)}} \right).$$

The theoretical convergence rates are $p+1$ for the L^2-norm and p for the energy norm.

Tables 4.6, 4.7, 4.8, 4.9 and 4.10 show the errors in L^∞, L^2 and H^1 norms and the empirical convergence rate. These test are made using a single GPU. The values of β_0 and β_1 are reported until they cannot decrease further. The minimum error reached depends on `toll` value utilized in the stop criterion shown in (A.3). In these tests, `toll=1e-15`.

Tables 4.11, 4.12, 4.13 and 4.14 show the empirical convergence rates for solution (E) computed using 4, 16, 64 and 256 GPUs. As usual, `toll=1e-15`.

h	p	β_1	$\|\|u-u_{ex}\|\|_{H^1}$	β_0	$\|\|u-u_{ex}\|\|_{L^2}$	$\|\|u-u_{ex}\|\|_{L^\infty}$
1/4	2		0.00739049		2.80407e-05	6.26224e-05
1/8	2	1.99219	0.00185765	3.94210	1.82432e-06	4.22253e-06
1/16	2	1.99778	0.000465128	3.97276	1.16193e-07	2.81756e-07
1/32	2	1.99943	0.000116328	3.95657	7.48402e-09	1.87392e-08
1/64	2	1.99986	2.90849e-05	3.86903	5.12202e-10	1.56137e-09
1/128	2	1.99996	7.2714e-06	3.64436	4.09619e-11	1.95847e-10
1/256	2	1.99999	1.81786e-06	3.33490	4.05954e-12	2.45233e-11
1/4	3		0.000286565		2.7659e-07	6.32347e-07
1/8	3	2.98981	3.60745e-05	4.95854	8.89545e-09	2.05703e-08
1/16	3	2.99745	4.5173e-06	4.97951	2.8196e-10	6.89143e-10
1/32	3	2.99936	5.64912e-07	4.97736	8.9506e-12	3.05925e-11
1/64	3	2.99984	7.06217e-08	4.96148	2.87275e-13	1.92761e-12
1/128	3	2.99996	8.82796e-09		2.63163e-14	1.20971e-13
1/256	3	2.99998	1.10351e-09		9.80976e-14	1.97509e-13
1/4	4		7.20957e-06		2.72374e-09	7.13528e-09
1/8	4	3.99045	4.5359e-07	5.94926	4.4082e-11	1.18966e-10
1/16	4	3.99761	2.83964e-08	5.93126	7.22392e-13	1.89762e-12
1/32	4	3.99940	1.77551e-09	5.39536	1.71636e-14	5.9946e-14
1/64	4	3.99985	1.10981e-10		4.57145e-14	9.24538e-14
1/128	4	3.98863	6.99123e-12		1.82945e-13	3.65249e-13
1/256	4		3.45527e-12		7.31723e-13	1.45788e-12
1/4	5		1.32429e-07		2.48461e-11	6.32797e-11
1/8	5	4.99106	4.16411e-09	6.95855	1.99768e-13	4.95784e-13
1/16	5	4.99776	1.3033e-10		3.8897e-15	1.0339e-14
1/32	5	4.99944	4.0744e-12		1.41046e-14	2.85189e-14
1/64	5		3.14038e-13		5.6419e-14	1.12757e-13
1/128	5		1.09468e-12		2.25813e-13	4.49307e-13
1/256	5		4.26335e-12		9.03371e-13	1.79555e-12
1/4	6		1.905e-09		2.04023e-13	5.31186e-13
1/8	6	5.99154	2.99408e-11	7.10486	1.48219e-15	4.31599e-15
1/16	6	5.99459	4.69583e-13		4.70708e-15	1.00059e-14
1/32	6		1.08565e-13		1.88896e-14	3.85247e-14
1/64	6		3.90506e-13		7.54868e-14	1.51698e-13
1/128	6		1.47438e-12		3.02087e-13	6.03018e-13
1/256	6		5.72227e-12		1.20812e-12	2.40671e-12

Table 4.6: Error norms of the Symmetric Interior Penalty Galerkin method for solution (A). We check the error norm rates β_0 (for L^2-norm) and β_1 (for H^1-norm) decreasing h.

h	p	β_1	$\|\|u - u_{ex}\|\|_{H^1}$	β_0	$\|\|u - u_{ex}\|\|_{L^2}$	$\|\|u - u_{ex}\|\|_{L^\infty}$
1/4	2		0.997578		0.00791414	0.0297358
1/8	2	1.96698	0.255168	3.84606	0.00055033	0.00233793
1/16	2	1.99153	0.0641676	3.94411	3.57543e-05	0.000162715
1/32	2	1.99786	0.0160657	3.97620	2.27181e-06	1.07844e-05
1/64	2	1.99946	0.00401792	3.96551	1.45423e-07	7.08997e-07
1/128	2	1.99987	0.00100457	3.89499	9.77519e-09	4.75003e-08
1/256	2	1.99997	0.000251149	3.69755	7.53447e-10	5.60675e-09
1/4	3		0.1149		0.000380506	0.000789292
1/8	3	2.95835	0.0147831	4.95485	1.22688e-05	2.85382e-05
1/16	3	2.98921	0.00186177	4.98601	3.87137e-07	9.22991e-07
1/32	3	2.99728	0.00023316	4.99404	1.21481e-08	3.29718e-08
1/64	3	2.99932	2.91588e-05	4.99483	3.80992e-10	1.2414e-09
1/128	3	2.99983	3.64528e-06	4.99176	1.19742e-11	7.43485e-11
1/256	3	2.99996	4.55673e-07		8.90226e-13	4.67332e-12
1/4	4		0.0100759		1.47519e-05	3.50703e-05
1/8	4	3.96602	0.000644754	5.95718	2.37441e-07	7.857e-07
1/16	4	3.99125	4.05423e-05	5.98447	3.75016e-09	1.31297e-08
1/32	4	3.99780	2.53777e-06	5.98115	5.93669e-11	2.09783e-10
1/64	4	3.99945	1.58671e-07	5.84103	1.03566e-12	3.98859e-12
1/128	4	3.99986	9.9179e-09		1.5047e-12	3.27871e-12
1/256	4	3.99622	6.21495e-10		6.01785e-12	1.28952e-11
1/4	5		0.000731747		5.63935e-07	1.29879e-06
1/8	5	4.97362	2.3289e-05	6.97321	4.48831e-09	1.20367e-08
1/16	5	4.99327	7.31186e-07	6.99053	3.52959e-11	9.71485e-11
1/32	5	4.99831	2.28764e-08	6.87772	3.00141e-13	8.3189e-13
1/64	5	4.99962	7.15075e-10		4.64032e-13	1.00431e-12
1/128	5	4.75219	2.65338e-11		1.85789e-12	3.99014e-12
1/256	5		5.60863e-11		7.43354e-12	1.59259e-11
1/4	6		4.59014e-05		2.20732e-08	5.02569e-08
1/8	6	5.97926	7.27594e-07	7.97313	8.78441e-11	2.59655e-10
1/16	6	5.99476	1.141e-08	7.97951	3.48048e-13	1.15907e-12
1/32	6	5.99865	1.78448e-10		1.55067e-13	3.53495e-13
1/64	6		5.85069e-12		6.19832e-13	1.34426e-12
1/128	6		1.93921e-11		2.48406e-12	5.35127e-12
1/256	6		7.523e-11		9.93255e-12	2.13067e-11

Table 4.7: Error norms of the Symmetric Interior Penalty Galerkin method for solution (B). We check the error norm rates β_0 (for L^2-norm) and β_1 (for H^1-norm) decreasing h.

h	p	β_1	$\|\|u-u_{ex}\|\|_{H^1}$	β_0	$\|\|u-u_{ex}\|\|_{L^2}$	$\|\|u-u_{ex}\|\|_{L^\infty}$
1/4	2		0.00403634		6.92595e-07	2.45032e-06
1/8	2	2	0.00100862	3.18034	7.64012e-08	3.06317e-07
1/16	2	2	0.000252148	3.11218	8.83569e-09	3.82904e-08
1/32	2	2	6.3037e-05	3.06380	1.05668e-09	4.78633e-09
1/64	2	2	1.57592e-05	3.03422	1.28989e-10	5.98292e-10
1/128	2	2	3.93981e-06	3.01772	1.59268e-11	7.47866e-11
1/4	3		3.73836e-14		1.54467e-15	6.2346e-15
1/8	3		3.39276e-13		1.0519e-14	5.8167e-14
1/16	3		4.05707e-13		1.00212e-14	4.58245e-14
1/32	3		9.6716e-13		2.9466e-14	8.92897e-14
1/64	3		2.36498e-12		7.34356e-14	2.30843e-13
1/128	3		5.46585e-12		1.75673e-13	6.27984e-13
1/4	4		1.94993e-13		5.98374e-15	2.71033e-14
1/8	4		5.3325e-13		1.73453e-14	8.79769e-14
1/16	4		1.15701e-12		3.48561e-14	1.72279e-13
1/32	4		1.61101e-12		4.89995e-14	1.43718e-13
1/64	4		4.47108e-12		1.47787e-13	5.23873e-13
1/128	4		1.11439e-11		4.37953e-13	1.61293e-12
1/4	5		3.49026e-13		1.31378e-14	4.56545e-14
1/8	5		6.23426e-13		1.76822e-14	8.25325e-14
1/16	5		7.9678e-13		2.10745e-14	9.60777e-14
1/32	5		3.31189e-12		1.0472e-13	3.40533e-13
1/64	5		6.49909e-12		2.20697e-13	6.63081e-13
1/128	5		1.74262e-11		6.83661e-13	2.10659e-12

Table 4.8: Error norms of the Symmetric Interior Penalty Galerkin method for solution (C). The polynomial degree of solution (C) equals 3. So, the numerical solution is exact for $p \geq 3$.

h	p	$\|\|u - u_{ex}\|\|_{H^1}$	$\|\|u - u_{ex}\|\|_{L^2}$	$\|\|u - u_{ex}\|\|_{L^\infty}$
1/4	2	1.31279e-14	7.58985e-16	2.08167e-15
1/8	2	8.31297e-13	3.33487e-14	1.37362e-13
1/16	2	6.65366e-13	2.04486e-14	1.05095e-13
1/32	2	2.99459e-12	9.93171e-14	5.81814e-13
1/64	2	6.77225e-12	2.40121e-13	1.19233e-12
1/128	2	1.01748e-11	3.86931e-13	1.82867e-12
1/4	3	3.79117e-13	1.55284e-14	4.99162e-14
1/8	3	7.93871e-13	2.31792e-14	1.113e-13
1/16	3	1.75366e-12	5.97531e-14	2.867e-13
1/32	3	3.9121e-12	1.3149e-13	6.53386e-13
1/64	3	1.09288e-11	4.25985e-13	1.95558e-12
1/128	3	2.56409e-11	1.14351e-12	3.76229e-12
1/4	4	4.27292e-13	1.26435e-14	7.55841e-14
1/8	4	1.10346e-12	3.5006e-14	1.58623e-13
1/16	4	3.16145e-12	1.1057e-13	5.38441e-13
1/32	4	8.09899e-12	3.0801e-13	1.54897e-12
1/64	4	1.69089e-11	7.24599e-13	2.82287e-12
1/128	4	4.00771e-11	1.84842e-12	4.72009e-12
1/4	5	8.31308e-13	2.74136e-14	9.88654e-14
1/8	5	1.8867e-12	6.37207e-14	2.25037e-13
1/16	5	6.19205e-12	2.43628e-13	1.0666e-12
1/32	5	1.16683e-11	4.69642e-13	2.00195e-12
1/64	5	3.06619e-11	1.38415e-12	4.0576e-12
1/128	5	5.96738e-11	2.66388e-12	7.24758e-12

Table 4.9: Error norms of the Symmetric Interior Penalty Galerkin method for solution (D). The polynomial degree of solution (D) equals 2. So, the numerical solution is exact for $p \geq 2$.

h	p	β_1	$\|\|u - u_{ex}\|\|_{H^1}$	β_0	$\|\|u - u_{ex}\|\|_{L^2}$	$\|\|u - u_{ex}\|\|_{L^\infty}$
1/4	2		100.257		18.1742	30.1482
1/8	2	2.52180	17.4574	6.35906	0.221405	0.530548
1/16	2	2.25613	3.65441	4.23671	0.0117439	0.0222896
1/32	2	2.00004	0.913573	4.10620	0.000681905	0.00154518
1/64	2	2.00003	0.228389	4.02269	4.19541e-05	9.91222e-05
1/128	2	2.00002	0.0570962	4.00076	2.62074e-06	6.23493e-06
1/256	2	2.00001	0.014274	3.98452	1.65564e-07	4.03133e-07
1/512	2	2	0.00356849	3.93856	1.0798e-08	2.68038e-08
1/4	3		26.3858		1.82256	3.08348
1/8	3	3.80212	1.89155	6.47512	0.0204869	0.0372516
1/16	3	2.03714	0.46087	5.51947	0.000446633	0.000906137
1/32	3	2.98186	0.0583376	5.04489	1.35297e-05	2.95833e-05
1/64	3	2.99547	0.00731516	5.00778	4.2053e-07	9.2824e-07
1/128	3	2.99887	0.000915112	5.00022	1.31395e-08	2.89468e-08
1/256	3	2.99972	0.000114411	4.99864	4.10998e-10	9.06419e-10
1/512	3	2.99993	1.43021e-05	4.95090	1.32883e-11	3.68355e-11
1/4	4		6.50855		0.126711	0.302995
1/8	4	2.82811	0.91651	7.78029	0.000576389	0.00122825
1/16	4	4.35744	0.0447112	5.01389	1.78395e-05	3.39006e-05
1/32	4	3.98013	0.00283321	5.99106	2.80476e-07	6.82419e-07
1/64	4	3.99506	0.000177683	5.99164	4.4079e-09	1.12879e-08
1/128	4	3.99877	1.11147e-05	5.98500	6.9593e-11	1.81133e-10
1/256	4	3.99969	6.94816e-07		2.64321e-12	9.73066e-12
1/512	4	3.99989	4.34292e-08		4.65543e-12	1.7137e-11
1/4	5		3.47723		0.0078239	0.013923
1/8	5	6.65787	0.0344363	6.00625	0.00012172	0.000221666
1/16	5	3.30179	0.00349203	7.42344	7.09061e-07	1.35223e-06
1/32	5	4.98083	0.000110586	6.99290	5.56685e-09	1.28886e-08
1/64	5	4.99521	3.4673e-06	6.99660	4.35937e-11	1.07234e-10
1/128	5	4.99880	1.08443e-07		1.71831e-12	6.10742e-12
1/256	5	4.99661	3.39682e-09		3.37022e-12	1.19817e-11
1/512	5		5.88574e-10		1.00817e-11	4.53131e-11
1/4	6		0.382002		0.00200687	0.00443417
1/8	6	4.30884	0.0192741	9.97592	1.99282e-06	3.75147e-06
1/16	6	6.40199	0.000227921	6.17062	2.76647e-08	5.16132e-08
1/32	6	5.98231	3.6052e-06	7.97537	1.09926e-10	2.72745e-10
1/64	6	5.99559	5.65038e-08		1.26413e-12	5.25224e-12
1/128	6	5.97722	8.96926e-10		2.31392e-12	7.59531e-12
1/256	6		4.43907e-10		7.57782e-12	3.25767e-11
1/512	6		8.26607e-10		1.43343e-11	6.52411e-11

Table 4.10: Error norms of the Symmetric Interior Penalty Galerkin method for solution (E). We check the error norm rates β_0 (for L^2-norm) and β_1 (for H^1-norm) decreasing h.

h	p	β_1	$\|\|u - u_{ex}\|\|_{H^1}$	β_0	$\|\|u - u_{ex}\|\|_{L^2}$
1/64	2		0.228389		4.19541e-05
1/128	2	2.00002	0.0570962	4.00076	2.62074e-06
1/256	2	2.00001	0.014274	3.98452	1.65564e-07
1/512	2	2	0.00356849	3.93855	1.0798e-08
1/64	3		0.00731516		4.2053e-07
1/128	3	2.99887	0.000915112	5.00022	1.31395e-08
1/256	3	2.99972	0.000114411	4.99866	4.10993e-10
1/512	3	2.99993	1.43021e-05	4.99792	1.28621e-11

Table 4.11: Error norms of the Symmetric Interior Penalty Galerkin method for solution (E) computed on 4 GPUs. We check the error norm rates β_0 (for L^2-norm) and β_1 (for H^1-norm) decreasing h.

h	p	β_1	$\|\|u - u_{ex}\|\|_{H^1}$	β_0	$\|\|u - u_{ex}\|\|_{L^2}$
1/128	2		0.0570962		2.62074e-06
1/256	2	2.00001	0.014274	3.98452	1.65564e-07
1/512	2	2	0.00356849	3.93855	1.0798e-08
1/1024	2	2	0.000892122	3.79554	7.77625e-10
1/128	3		0.000915112		1.31395e-08
1/256	3	2.99972	0.000114411	4.99866	4.10993e-10
1/512	3	2.99993	1.43021e-05	4.99792	1.28621e-11
1/1024	3	2.99998	1.78779e-06	4.74136	4.80861e-13

Table 4.12: Error norms of the Symmetric Interior Penalty Galerkin method for solution (E) computed on 16 GPUs. We check the error norm rates β_0 (for L^2-norm) and β_1 (for H^1-norm) decreasing h.

h	p	β_1	$\|\|u-u_{ex}\|\|_{H^1}$	β_0	$\|\|u-u_{ex}\|\|_{L^2}$
1/256	2		0.014274		1.65564e-07
1/512	2	2	0.00356849	3.93855	1.0798e-08
1/1024	2	2	0.000892122	3.79555	7.77623e-10
1/2048	2	2	0.00022303	3.469	7.02257e-11
1/256	3		0.000114411		4.10994e-10
1/512	3	2.99993	1.43021e-05	4.99792	1.28621e-11
1/1024	3	2.99998	1.78779e-06	4.74178	4.8072e-13
1/2048	3	3	2.23473e-07		1.04887e-12

Table 4.13: Error norms of the Symmetric Interior Penalty Galerkin method for solution (E) computed on 64 GPUs. We check the error norm rates β_0 (for L^2-norm) and β_1 (for H^1-norm) decreasing h.

h	p	β_1	$\|\|u-u_{ex}\|\|_{H^1}$	β_0	$\|\|u-u_{ex}\|\|_{L^2}$
1/512	2		0.00356849		1.0798e-08
1/1024	2	2	0.000892122	3.79555	7.77624e-10
1/2048	2	2	0.00022303	3.46901	7.02252e-11
1/4096	2	2	5.57576e-05		4.25274e-11
1/512	3		1.43021e-05		1.28621e-11
1/1024	3	2.99998	1.78779e-06	4.73576	4.8273e-13
1/2048	3	3	2.23473e-07		1.04808e-12
1/4096	3	2.99998	2.79344e-08		4.19486e-12

Table 4.14: Error norms of the Symmetric Interior Penalty Galerkin method for solution (E) computed on 256 GPUs. We check the error norm rates β_0 (for L^2-norm) and β_1 (for H^1-norm) decreasing h.

4.4 Kernel time complexity

We have verified the time complexity of the kernels varying the number of mesh elements and the basis degree. We compute the empirical time complexity as

$$\begin{aligned} \zeta_p &= \log_2\left(\frac{\tau(noe,p)}{\tau(noe,p/2)} \right), \\ \zeta_{noe} &= \log_4\left(\frac{\tau(noe,p)}{\tau(noe/4,p)} \right), \end{aligned}$$

where $\tau(noe,p)$ is the kernel execution time for noe mesh elements with p-degree basis functions.

The theoretical time complexity as a function of noe is linear for all the kernels, i.e. $O(noe)$. The theoretical time complexity as a function of the basis degree p is cubic for `volume_mvm< ,0>` and `volume_mvm< ,0>` (i.e. $O(p^3)$), biquadratic for `volume` (i.e. $O(p^4)$) and quadratic (i.e. $O(p^2)$) for `flux6a` and `flux6b`.

Tables 4.15 and 4.16 show kernels execution times and their empirical time complexities. The `flux` column shows the sum of the execution times of `flux6a` and `flux6b` kernels. The results are satisfactory because the empirical time complexity does not exceed the theoretical time complexity. These tests have been run on a Tesla K20x GPU.

mesh size	volume_mvm$<,1>$		volume_mvm$<,0>$		volume		flux	
	ζ_{noe}	time	ζ_{noe}	time	ζ_{noe}	time	ζ_{noe}	time
128×128		0.147266		0.334439		0.243877		0.157058
256×256	1.2218	0.801135	1.0687	1.471548	0.9959	0.970066	0.9735	0.605640
512×512	0.9954	3.184381	1.0017	5.900559	0.9933	3.844745	1.0019	2.429092
1024×1024	1.0006	12.74895	0.9995	23.58595	0.9990	15.35866	1.0248	10.05710
2048×2048	1.0463	54.38087			0.9991	61.36535	1.0006	40.26576

Table 4.15: Execution times varying the mesh size with $p = 4$. Times in ms.

p	volume_mvm$<,1>$		volume_mvm$<,0>$		volume		flux	
	ζ_p	time	ζ_p	time	ζ_p	time	ζ_p	time
2		0.682022		1.222571		0.709510		0.904528
4	2.2295	3.198525	2.2742	5.914229	2.4397	3.849379	1.4276	2.433197
8	2.5206	18.35405	2.5577	34.82236	3.1662	34.55660	1.6920	7.861967
16	2.7733	125.4853			3.5754	411.9650	1.8243	27.84265
32	2.8357	895.8504			3.8204	5820	1.8706	101.8167

Table 4.16: Execution times varying p, the mesh size is 512×512. Times in ms.

4.5 Kernel bandwidth

Table 4.17 shows the bandwidth usage of the kernels: `volume`, `flux6a` and `flux6b`. We have measured the bandwidth (using the NVIDIA Visual Profiler) increasing the polynomial degree of the basis functions on a 512×512 mesh. These measures have been carried out on a Tesla K20x GPU, which have a maximum bandwidth of 250 GB/s.

p	`volume`	`flux6a`	`flux6b`
2	194.7 GB/s	182.9 GB/s	191.5 GB/s
4	171.1 GB/s	189.4 GB/s	194.6 GB/s
8	112.5 GB/s	189.0 GB/s	194.4 GB/s

Table 4.17: Kernel bandwidth.

4.6 Occupancy

Table 4.18 shows the theoretical occupancy (see Section 1.4.1) of `volume`, `flux6a` and `flux6b` kernels. These theoretical occupancy percentages are associated to compute capability 3.5. Moreover, the block size of these kernels is set to 128 threads. The effective occupancy (σ) is very close to the theoretical occupancy (σ_t), usually $\sigma_t - \sigma < 3\%$.

	occupancy
`volume`	100%
`flux6a`	62.5%
`flux6a`	62.5%

Table 4.18: Kernel theoretical occupancy.

4.7 Multi-GPU scalability

In this section we investigate the multi-GPU scaling efficiency. We have measured the execution times of the matrix-vector product (which has been performed launching the `volume`, `flux6a` and `flux6b` kernels) increasing the number of GPUs connected in parallel with MPI. These measures have been carried out on the CSCS Piz Daint system.

We have restricted the assessment of the code parallel performance to this simple algebraic operation for two reasons. On one hand, it is the key step in any iterative solver, so that its performance should follow closely that of the resulting method. On the other hand, in the present preliminary implementation, the lack of an advanced parallel preconditioner affects negatively the overall parallel performance of the solver we have developed.

There are two ways to evaluate the performance of a parallel code: strong and the weak scaling. Strong scaling means measuring the execution times for a fixed size problem while increasing the number of GPUs. As a consequence, the problem size assigned to each GPU decreases. The strong scaling efficiency ε_S is defined as

$$\varepsilon_S(n) = \frac{\tau(n)}{n\ \tau(1)} ,$$

where n is the number of GPUs and $\tau(n)$ is the execution time for a fixed size problem. A code is considered to scale linearly if ε_S is equal to 1. The ratio $\tau(n)/\tau(1)$ is called speedup. In weak scaling, the problem size assigned to each GPU stays constant, so that using additional GPUs makes the size of the problem grow. The weak scaling efficiency ε_W is defined as

$$\varepsilon_W(n) = \frac{\bar{\tau}(n)}{\bar{\tau}(1)} ,$$

where n is the number of GPUs and $\bar{\tau}(n)$ is the execution time for problems of size increasing with n.

Table 4.19 shows ε_W values for the weak scaling. These values highlight that the execution times depend mostly from the size of the problem assigned to each GPU. If the local problem is big enough, the communication costs between GPUs have a lower impact on the overall computational cost. In case of $p = 8$, the ε_W values, for a mesh assigned to each GPU larger than 256×256, are high and they stay constant adding more GPUs. Whereas, if the mesh assigned to each GPU is smaller than 256×256, the ε_W values are lower and they decrease adding more GPUs.

Tables 4.20 and 4.21 show respectively the strong scaling of a 1024×1024 mesh with $p = 8,\ 16$ and of a 2048×2048 mesh with $p = 4,\ 8$. Figure 4.4 displays the ε_S values. As expected, we notice that a larger mesh and a higher basis degree produce higher ε_S values. In order to have a strong scaling efficiency, it is important to maximize the size of the local problem being solved by each GPU.

GPUs	128×128 time	ε_W	256×256 time	ε_W	512×512 time	ε_W	1024×1024 time	ε_W	2048×2048 time	ε_W
1	2.65282		10.5073		42.1191		169.081		676.460	
4	3.04866	0.87	11.3461	0.93	44.2908	0.95	176.567	0.96	703.540	0.96
16	3.24599	0.82	11.7559	0.89	44.9849	0.94	177.998	0.95	706.181	0.95
64	3.27948	0.81	11.8418	0.89	45.0136	0.93	178.817	0.94	719.029	0.94
256	3.47229	0.76	11.8125	0.89	45.2128	0.93	178.900	0.94	709.932	0.95
1024	3.38314	0.78	11.9889	0.88	45.5493	0.92	179.773	0.94	713.466	0.95

Table 4.19: Weak scaling for $p = 8$. Times in ms.

	$p = 8$			$p = 16$		
GPUs	time	speedup	ε_S	time	speedup	ε_S
1	169.081			1784.95		
4	44.2908	3.82	0.95	455.501	3.92	0.98
16	11.7559	14.4	0.90	115.909	15.4	0.96
64	3.27948	51.6	0.81	30.2832	58.9	0.92

Table 4.20: Strong scaling for a 1024×1024 mesh. Times in ms.

	$p = 4$			$p = 8$		
GPUs	time	speedup	ε_S	time	speedup	ε_S
1	100.544			676.460		
4	27.6949	3.63	0.91	176.567	3.83	0.96
16	7.16698	14.0	0.87	44.9849	15.0	0.94
64	2.07761	48.4	0.76	11.8418	57.1	0.89

Table 4.21: Strong scaling for a 2048×2048 mesh. Times in ms.

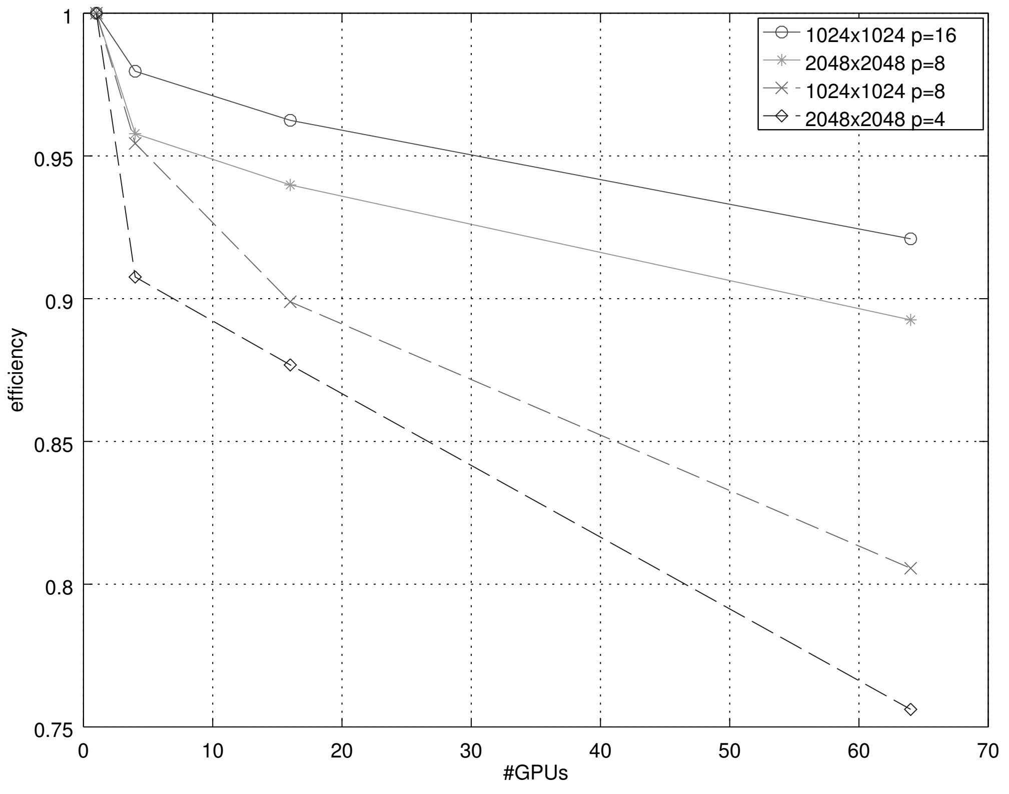


Figure 4.4: Strong scaling efficiency plot.

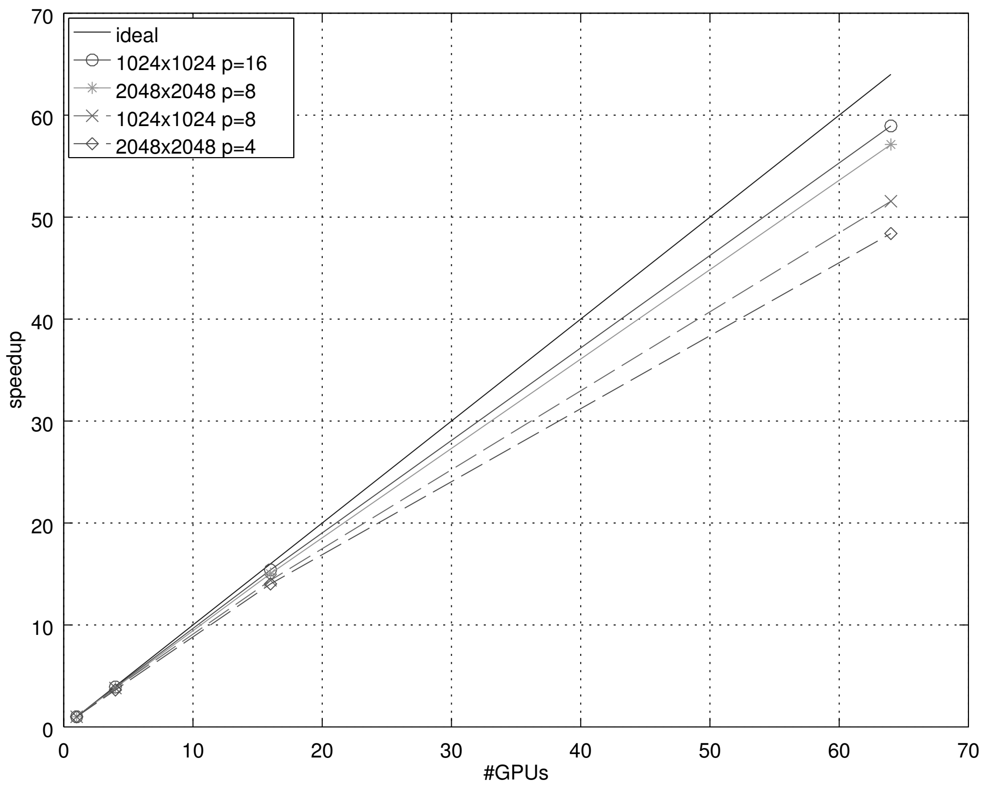


Figure 4.5: Strong scaling speedup plot.

Conclusions

In this thesis, we have developed a code which solves a Poisson problem with Dirichlet boundary conditions. The problem has been discretized with the Symmetric Interior Penalty Galerkin method with high order basis functions. The linear system obtained has been solved on GPUs. In particular, we have implemented a stiffness matrix-vector product function which works on one or more GPUs. This function does not use global stiffness matrix entries stored in memory, but it recompute them at each iteration, in order to avoid the large latency of global memory transactions.

The code has been developed in the CUDA programming language and first validated with respect to the correctness of the implementation. Subsequently, performance analyses were carried out for both the single GPU and multi-GPU versions. The developed kernels reach a high occupancy and exploit most of the available bandwidth.

Strong scaling results for the multi-GPUs code appear satisfactory, at least for problems of sufficiently large size with respect to the number of available GPUs.

Future developments of this thesis work include extending the code to three dimensional problem on fully unstructured meshes, as well as the extension of the parallel implementation to a multi-CPU/multi-GPU code able to exploit fully the computational potential of hybrid parallel architectures.

Appendix A

Basic results in numerical analysis

In this chapter, we present some basic results in the numerical analysis of finite element methods which have been implemented in this thesis. The results of the first two section have been taken from [9]. The Conjugate Gradient method algorithm has been taken from [11].

A.1 Legendre polynomials

A Legendre polynomial $L_k : [-1, 1] \to \mathbb{R}$ is the solution of the Legendre differential equation

$$\left[(1-x^2)L_k'(x)\right]' + k(k+1)L_k(x) = 0 \qquad -1 \le x \le 1 \tag{A.1}$$

for a given $k \in \mathbb{N}$. The solution of equation (A.1) is a polynomial in which the total degree equals k, which is an even (odd) function if k is even (odd).

The values of L_k and L_k' at a point x can be computed using the following recursion relations:

$$\begin{aligned} L_k(x) &= \frac{2k-1}{k}\ x\ L_{k-1}(x) - \frac{k-1}{k}\ L_{k-2}(x) \\ L_k'(x) &= \frac{k(k+1)}{(2k+1)(1-x^2)}\ \left[\ L_{k-1}(x) - L_{k+1}(x)\ \right] \end{aligned}$$

where $L_1(x) = x$ and $L_0(x) = 1$. Moreover the second order derivative can be directly obtained from equation (A.1). The values of such functions at the end points are:

$$\begin{aligned} L_k(\pm 1) &= (\pm 1)^k \\ L_k'(\pm 1) &= \frac{1}{2}(\pm 1)^{k-1}k(k+1) \\ L_k''(\pm 1) &= \frac{1}{8}(\pm 1)^k(k-1)k(k+1)(k+2) \end{aligned}$$

Legendre polynomials are orthogonal, i.e.

$$\int_{-1}^{1} L_m(x)\ L_n(x)\ dx \ = \ \frac{2}{2n+1}\ \delta_{nm}$$

where δ_{nm} is the Dirac delta function, which equals 1 if $m = n$ otherwise it is zero.

A.2 Gauss Legendre Lobatto quadrature

The quadrature of a function is the approximation of its definite integral on a given domain. In our code, we employ the Gauss-Legendre-Lobatto quadrature rule.

Let $f : [-1, 1] \to \mathbb{R}$ be a function well-approximated by polynomials, its quadrature can be expressed as a weighted summation of its values at specified points.

$$\int_{-1}^{1} f(x) \approx \sum_{i=0}^{N} w_i f(x_i) \tag{A.2}$$

where $N + 1$ is the number of quadrature points, w_i are the weights and x_i are the nodes.

The nodes include the end-points of the domain (± 1). Gauss Legendre Lobatto quadrature rule is accurate for polynomials up to degree $2N - 1$.

The quadrature nodes are the zeros of $(1 - x^2)L'_N(x)$. We compute these values using the Newton method

$$x_i^{k+1} = x_i^k - \frac{L'_N(x_i^k)}{L''_N(x_i^k)} \quad k \geq 0$$

where $1 \leq i \leq N - 1$ because $x_0 = -1$ and $x_N = 1$.

There is one zero of $L'_N(x)$ between two consecutive zeros of $L_N(x)$. So, we compute the initial guess x_i^0 as

$$x_i^0 = \frac{\sigma_j + \sigma_{j+1}}{2} \quad 1 \leq j \leq N - 1$$

where σ_j is the approximation of j^{th} zero of L_{n-1} which is calculated as

$$\sigma_k = \Big[1 - \frac{N-1}{8N^3} - \frac{1}{384N^4}\Big(39 - \frac{28}{\sin^2\theta_k}\Big)\Big]\cos\theta_k$$

$$\theta_k = \frac{4k-1}{4N+2}\pi \quad 1 \leq k \leq N$$

The quadrature weights are

$$w_i = \frac{2}{N(N+1)}\frac{1}{L_N^2(x_i)}$$

The integrations on arbitrary domains are performed by scaling results of integrals on $[-1, 1]$.

$$\int_a^b f(x)\, dx = \frac{b-a}{2}\int_{-1}^{1} f\Big(\, \frac{b-a}{2}z + \frac{a+b}{2}\, \Big)\, dz$$

A.3 Conjugate Gradient method

The linear systems have been solved with an iterative method. In this section we introduce the algorithm which we have implemented in our code.

Let $\mathbf{A}$ be a symmetric and positive definite matrix. The linear system

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

can be solved using the Conjugate Gradient method [11], which reads as follow.

```
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 = \mathbf{r}_0$
$k = 0$
while $r_k \neq 0$
    $\alpha_k = \frac{\mathbf{r}_k^{\mathsf{T}}\mathbf{r}_k}{\mathbf{p}_k^{\mathsf{T}}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k \mathbf{A}\mathbf{p}_k$
    $\beta_k = \frac{\mathbf{r}_{k+1}^{\mathsf{T}}\mathbf{r}_{k+1}}{\mathbf{r}_k^{\mathsf{T}}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
    $k = k + 1$
end while
```

When iterations stop, vector $\mathbf{x}_k$ contains the linear system solution. The condition $\mathbf{r}_k \neq 0$ cannot be reached in the reality. So, the iterations are stopped when

$$\frac{||\mathbf{r}_k||}{||\mathbf{b}||} < \texttt{toll} \tag{A.3}$$

where $||\cdot||$ is the Euclidean norm and `toll` is a small enough value.

Bibliography

[1] P.F. Antonietti, M. Sarti, M. Verani (2015). *Multigrid algorithms for hp-Discontinuous Galerkin discretizations of elliptic problems.*SIAM Journal on Numerical Analysis, 53(1), 598-618.

[2] S. Salsa (2010). *Equazioni a derivate parziali: Metodi, modelli e applicazioni.* UNITEXT, Springer Italia.

[3] A. Quarteroni (2012). *Modellistica Numerica per Problemi Differenziali, 5a edizione.* UNITEXT, Springer-Verlag Italia.

[4] D.A. Di Pietro, A. Ern (2012). *Mathematical Aspects of Discontinuous Galerkin Methods.* Mathématiques et Applications 69, Springer-Verlag Berlin Heidelberg.

[5] B. Riviere (2008). *Discontinuous Galerkin Methods For Solving Elliptic And parabolic Equations: Theory and Implementation.* Society for Industrial and Applied Mathematics, Philadelphia.

[6] M. Bianco, U. Varetto (2012). *A generic library for stencil computations.* arXiv preprint arXiv:1207.1746.

[7] C. Canuto, Y. Hussaini, A. Quarteroni, T. Zang (2006). *Spectral Methods: Fundamentals in Single Domains.* Springer.

[8] C. Canuto, Y. Hussaini, A. Quarteroni, T. Zang (2007). *Spectral Methods: Evolution to Complex Geometries and Applications to Fluid Dynamics*, Springer.

[9] J. Shen, T. Tang, L. Wang (2008). *Spectral Methods: Algorithms, Analysis and Applications*, Springer.

[10] J. Freund, R. Stenberg (1995). *On weakly imposed boundary conditions for second order problems.* Proceedings of the International Conference on Finite Elements in Fluids.

[11] J. Nocedal, S. Wright (2006). *Numerical Optimization.* Springer.

[12] A. Klöckner, T. Warburton, J. Bridge, J.S. Hesthaven (2009). *Nodal discontinuous Galerkin methods on graphics processors.* Journal of Computational Physics, 228(21), 7863-7882.

[13] M. Naumov (2011). *Incomplete-LU and Cholesky Preconditioned Iterative Methods Using CUSPARSE and CUBLAS.* CUDA Toolkit Documentation.

[14] C. Cecka, A.J. Lew, E. Darve (2011). *Assembly of finite element methods on graphics processors.* International journal for numerical methods in engineering, 85(5), 640-669.

[15] G.R. Markall, A. Slemmer, D.A. Ham, P.H.J. Kelly, C.D. Cantwell, S.J. Sherwin (2013). *Finite element assembly strategies on multi-core and many-core architectures.* International Journal for Numerical Methods in Fluids, 71(1), 80-97.

[16] J. Nickolls, I. Buck, M. Garland, K. Skadron (2008). *Scalable Parallel PROGRAMMING with CUDA.* ACM Queue, 6(2), 40-53.

[17] R. Farber (2011). *CUDA Application Design and Development*, Elsevier.

[18] CUDA toolkit `https://developer.nvidia.com/cuda-toolkit`

[19] *CUDA C programming guide*, `http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

[20] *CUDA C best practices guide*, `http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html`

[21] V. Volkov (2010). `http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf` , slides.

[22] *NVIDIA Fermi Architecture Whitepaper.* `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`

[23] *NVIDIA Kepler GK110 GK210 Architecture Whitepaper.* `http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf`

[24] *Thrust library.* `http://docs.nvidia.com/cuda/thrust/`