Books on parallel programming theory often talk about such weird beasts like the PRAM model, a hypothetical hardware that would provide the programmer with a number of processors that is proportional to the input size of the problem at hand. Modern general purpose computers afford only a few processing units; four is currently a reasonable number. This limitation makes the development of highly parallel applications quite difficult to the average computer user. However, the low cost and the increasing programmability of graphics processing units, popularly known as GPUs, is contributing to overcome this difficulty. Presently, the application developer can have access, for a few hundred dollars, to a hardware boosting hundreds of processing elements. This brave new world that is now open to many programmers brings, alongside the incredible possibilities, also difficulties and challenges. Perhaps, for the first time since the popularization of computers, it makes sense to open the compiler books on the final chapters, which talk about very unusual concepts, such as polyhedral loops, iteration space and Fourier-Motskin transformations, only to name a few of these chimerical creatures. This material covers, in a very condensed way, some code generation and optimization techniques that a compiler would use to produce efficient code for graphics processing units. Through these techniques, the compiler writer tries to free the application developer from the intricacies and subtleties of GPU programming, giving him more freedom to focus on algorithms instead of micro-optimizations. We will discuss a little bit of what are GPUs, which applications should target them, how the compiler sees a GPU program and how the compiler can transform this program so that it will take more from this very powerful hardware.
1 The Rise of GPU computing

The objective of this section is to explain what is the hardware for which we will be generating code. We will answer questions like:

- what are graphics processing units;
- which applications most benefit from GPUs;
- what are SIMD machines and what this concept has to do with GPUs;
- how to port my C program to run on a GPU;

**Fiat Lux:** Most of the computers that we buy on the shops come with a graphics card. This very specialized hardware is responsible for rendering the beautiful images that you can display on your screen – after all, who draws all those bullets that you fire from your machine gun when you are playing *Halo*? Normally we cannot use these processors to run the programs that we write, say, in C, Java, or the APL one-liners – in case you are a masochist. The graphics card usually comes with a few hard-coded software that help the Central Processing Unit (CPU) to deal with graphic intensive applications. That is a pity, for graphics hardware generally are very parallel.

**Question 1.1** Why would one expect that the graphics card be massively parallel?

However, things are changing. Nowadays it is possible for hoi polloi 2 like me to buy a programmable graphics board.

**Question 1.2** 15 years ago nobody would talk about GPUs. Instead, PCs would rely on Video Graphics Arrays (VGA) for graphics computing. What is VGA?

The evolution from Video Graphics Arrays to GPUs was gradual. Initially VGAs were augmented with extra hardware to do rasterization, texture mapping and shading.

**Question 1.3** Rasterization, texture mapping and shading are part of the typical jargon of the graphics programmer. What are these concepts?

At this point, the graphics card was more than a simple VGA, for it had some sort of a processor, albeit a very specialized one. That was when the graphics stuff got the “P” in “GPU”. The next step into this evolution was the increase in programability. Fixed logic was replaced by floating-point units, and later double precision floating point arithmetics. From this stage to processing instructions, private memory and a less specialized pipeline was piece of candy.

**Question 1.4** So, today we have a “programmable GPU”. They do not come in every computer. What are the steps to have access to one?

---

2Words like “hoi polloi” are what you profit from the GRA
Johnny Mnemonic: Johnny Mnemonic was the first very popular character idealized by William Gibson, the father of the cyberspace. Johnny was an “information dealer”. He had a second brain, where he would store information that he would transport for eventual customers. Johnny had a heterogeneous nervous system, much in the same way that modern computers start to bear heterogenous processors.

A GPU has way more processing units than an ordinary CPU. However, these processing elements are more specialized, and generally slower than the general purpose hardware that one is likely to find on a modern CPU.

Question 1.5 Given these differences between GPUs and CPUs, which are the applications that are likely to benefit the most from a GPU?

If you answered “data parallel” applications, then you certainly know what you are talking about. Indeed, although one can, in principle, run anything on a GPU, the applications that really shine in this environment are the very regular programs. In this category of software we find a lot of linear algebra, image processing and video encoding/decoding problems. And, of course, a lot of graphics processing, both 2D and 3D. On the other hand, interactive applications, that is, those that do a lot of talking with the user, fare better on the CPU.

Question 1.6 Ok, so we have applications that do better on the GPU, and applications that should run on the CPU instead. What is the natural thing to do?

Heterogeneous hardware is not exactly a new trend. Massively parallel computers have been using SIMD accelerators since forever; however, these were not very popular machines. CPU-GPU combinations are changing this landscape. Figure 1 compares a traditional VGA-based architecture with the heterogenous GPU-based hardware.

Figura 1: Traditional versus heterogenous graphics hardware.

Question 1.7 There are other examples of heterogenous processors, even in low-cost PCs. Can you think about other examples?

Question 1.8 Who decides which tasks run on the CPU, and which tasks are sent to the GPU?
Set, Aim, Fire! To be part of a French fire squad was no easy task. That is, shooting at point-blank range a target tied till the core of the bones was not the hard part. The problem was the guilt that would come after the bullet flew into its destination. Because of this, the four-man party would receive one gun loaded with phony bullets, so that each officer could believe that his was not the powder to blame. The fire square is a good analogy to a GPU hardware.

The GPU is quite an amazing piece of hardware. Quoting the Nvidia Best Practices Guide [4]: “all Nvidia GPUs can support 768 active threads per multiprocessor, and some GPUs support 1,024 active threads per multiprocessor. On devices that have 30 multiprocessors (such as the NVIDIA GeForce GTX 280), this leads to more than 30,000 active threads.” Such a hardware has allowed GPU based applications to run over 400x faster than corresponding CPU based programs [33]. Figure 2 shows a typical blueprint of a GPU, compared to a CPU. Notice that the GPU chip separates a much greater proportion of its area to processing elements than the CPU.

**Question 1.9** Why GPUs are so parallel? And why traditional CPUs do not show off all this parallelism? Would it not be very good if our ordinary PC had dozens of processing units?

![Figura 2: A comparison between the CPU and the GPU micro-chip, taken from the CUDA programming manual [3].](image)

Some authors say that the GPU follows the Single Instruction Multiple Thread \textit{SIMT} execution model [29]. However, this terms is not really standard; it was coined by some Nvidia engineers, not too long ago. If one had to follow Flynn’s taxonomy [16], we could say that GPUs follow the MSIMD model. This acronym stands for (Multiple) Single Instruction Multiple Data. Put in other words, the GPU has a number of processors, which execute independently – from where we have the “Multiple” in MSIMD. Each individual processor, on its turn, is a Single Instruction Multiple Data (SIMD) machine.

**Question 1.10** \textit{SIMD, SPMD...}, what else can we find in Flynn’s taxonomy?
In my opinion, the best way to understand how the SIMD hardware works is to compare it with the fire squad. Let’s assume that the fire squad is made of four cold-blooded soldiers and a very vocal captain. The captain knows how to scream three words: “set”, “aim” and “fire”. Upon hearing each of these words, the four soldiers take a different suite of actions. For instance, if the soldiers hear “fire”, then all of them pull the trigger together; however, they all use different guns. That is: this instruction “fire”, applies to every soldier, but the soldiers use different guns and bullets.

**Question 1.11 How can we transpose captain, soldiers, guns (and roses) to the SIMD world?**

A good one now is the gun with the fake bullets. You know: in this way each shooter can sleep better, believing that his was not the gun that nuked the doomed chest away.

**Question 1.12 Before moving on to the next paragraph, think a little bit: what the phony gun has to do with SIMD execution?**

The answer, in this case, is *conditional execution*. The SIMD hardware contains dozens of processing elements, and they all execute the same instruction each time. However, sometimes it is interesting to turn off some of these processors. There are many ways to do this. A typical approach is to *predicate* the instructions that are sent to each processor. If a given predicate is true, then execute the instruction, otherwise just do a no-op. We will explain the SIMD model a bit better later on, when we hope that uncanny goblins such as conditional execution will be made more clear. This changeling, in particular, will be quite important to us in this course.

**Question 1.13 Can you think about the kind of hardware necessary to implement conditional execution?**

**Man plus:** It is natural that a heterogenous hardware would ask for a heterogenous programming language. There are, in fact, many programming languages that have been developed to program this kind of hardware. Some languages are very domain specific, i.e, they are used mostly for graphics programming, like Cg (C for graphics) or HLSL. Cg, for instance, is basically a collection of functions that can be invoked on a slightly different dialect of C. Other languages are more general; perhaps the most popular member in the latter category is C for CUDA.

**Question 1.14 What do you think a language like C for CUDA should have? What are the main abstractions that this language should provide?**

C for CUDA has a very fundamental difference from traditional programming languages such as C, C++, Java, Php and the wonderful Linf [31]. These languages are meant to run in any general purpose hardware. More yet, they tend to abstract the hardware away. By hiding from the programmer hardware details the programming language effectively foster
the development of more portable, and generally more readable code. It is not very easy to abstract heterogeneity away... not that it may not happen in the future, after all, programmers always want more abstractions, and compiler writers seem to live to satisfy their needs. However, when programming in C for CUDA the developer knows precisely which parts of the code are meant to run on the CPU, and which parts will go to the GPU.

**Question 1.15** Heterogeneity is not the only aspect of the hardware that C for CUDA lets surface to the programmer. Which other forms of “hardware semantics” are visible in C for CUDA?

**A hybrid programming language:** C for CUDA relies on three main abstractions:

- a hierarchy of threads;
- shared memory;
- barrier synchronization.

A CUDA program contains a sequential body, written in C, plus one or more kernels. A kernel is a program that will run in the GPU. Looking at a kernel we have the impression that it is a sequential program. However, that code will be executed by dozens of threads at the same time. CUDA organizes these threads in a hierarchy of thread blocks and thread grids.

- Threads inside the same block follow the Single Program Multiple Data (SPMD) execution model. These threads can communicate via fast shared memory, and can synchronize via barriers.
- Threads inside the same grid can cooperate via the slow global memory. However, threads in different grids execute independently.

**Question 1.16** Why do we have this hierarchy? What is the purpose of organizing threads in blocks and grids?

**Question 1.17** In terms of programming languages, what distinguishes threads inside the same block or grid?

To answer the last question, we notice that threads have unique identifiers, which determine the thread inside a block, and the block inside a grid. Let’s illustrate this with a simple program that computes the vectorial operation \( y = \alpha x + y \) for vectors \( x \) and \( y \). The sequential C program is given in Figure 3, whereas the parallel program written in C for CUDA appears in Figure 4.

**Question 1.18** In Figure 4 we divide the threads into blocks. Can you simplify the program to use only one thread block?
void saxpy_serial(int n, float alpha, float *x, float *y) {
    for (int i = 0; i < n; i++)
        y[i] = alpha*x[i] + y[i];
}

// Invoke the serial function:
saxpy_serial(n, 2.0, x, y);

Figura 3: C program that computes $y = Ax + y$, taken from Nickolls and Kirk [29].

__global__
void saxpy_parallel(int n, float alpha, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) y[i] = alpha * x[i] + y[i];
}

// Invoke the parallel kernel:
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);

Figura 4: CUDA program that computes $y = Ax + y$, taken from Nickolls and Kirk [29].

**Question 1.19** What is the relation between the number of processors and the number of threads? Can we have more threads than processing elements in the GPU?

**Question 1.20** A curious limitation in the original CUDA programming model is the absence of recursion. Why C for CUDA would not support it? And how to program usually recursive algorithms, like quicksort?

**Electronic Cinderella** Many CUDA programs are direct translations of programs originally developed to run on sequential machines. This translation can be done mechanically, via compilation techniques known since long ago [23]. The translation algorithms rely on a vast linear algebra theory, using concepts such as vectorial spaces and affine transforms. Here we will sweep all of this under the carpet, keeping, nevertheless, the concept of *iteration space*, and using a good deal of intuition.

The iteration space of a loop is the set formed by every possible combination of indices in the loop. Consider, for instance, the C program in Figure 5, which adds up two matrices together.

If we assume that \texttt{side} = 10, then the iteration space of this application is given in Figure 6. We shall stick to \texttt{side} = 10 in the next examples. This figure describes the best scenario for parallelism: there is no temporal dependencies between any pair of points in the space. That is, on a PRAM model, we could assign each sum $A_{i,j} + B_{i,j}$ to a processor, and we could perform the whole summation in constant time. Coincidentally, we could do something very similar in CUDA.
void matSum(float** s, float** a, float** b, unsigned int side) {
    int i, j;
    for (i = 0; i < side; i++) {
        for (j = 0; j < side; j++) {
            s[i][j] = a[i][j] + b[i][j];
        }
    }
}

Figura 5: C program that computes the matrix sum $S = A + B$.

Figure 6: Iteration space for the program in Figure 5.

Figure 7 shows the kernel that is responsible for performing this product. The original distribution of C for CUDA did not support multidimensional arrays, so, normally programmers linearize arrays that have more than one dimension. This is nothing more than doing, at the source code level, what the compiler already does under the hood. After all, how do you think gcc represents matrices?

The program in Figure 5 was the perfect scenario for parallelization: no dependencies between any iteration of the loop. This case is more common than one would at first
__global__ void matSumKernel(float* S, float* A, float* B, int side) {
    int i = blockIdx.y*blockDim.y + threadIdx.y;
    int j = blockIdx.x*blockDim.x + threadIdx.x;
    int ij = i*side + j;
    if (ij < side*side) {
    }
}

Figura 7: The CUDA program that is equivalent to the C program in Figure 5

void depSum(float** s, unsigned int side) {
    int i, j;
    for (i = 0; i < side; i++) {
        for (j = 1; j < side; j++) {
            s[i][j] = s[i][j] - s[i][j-1];
        }
    }
}

Figura 8: C program illustrating temporal dependencies.

guess. Many applications are like this; however, we also have many applications that present dependencies. Let’s consider, for instance, the program in Figure 8, and its corresponding iteration space, in Figure 9. The arrows show the direction of dependences between iterations. As we can see by looking at the plot, this program is not as parallel as that in Figure 5, for iteration \((i, j)\) depends on iteration \((i, j - 1)\). On a PRAM model, we could assign each line to a processor; hence, converting a problem which has quadratic complexity into a problem that has linear solution, with no additional increase in the amount of work that is performed by the parallel implementation.

Figure 10 shows the CUDA program that corresponds to the program in Figure 8. This program has linear complexity. Intuitively, we can imagine that the geometric representation of the iteration space describes a graph, where iteration dots are vertices, and there exists an edge from vertex \(v_1\) to vertex \(v_2\) if, and only if, \(v_1\) depends on \(v_2\). We can trivially parallelize the loop thus represented by assigning an individual process to each connected component in this graph. In our simple example, each connected component is a line, and our parallelization approach will demand as many processing elements as there are lines in our chart.

In the two examples seen so far, all the processors in the parallel solution perform the same amount of work. That is, in a perfect SIMD world, where we have as many processors as independent work to be done, all the processors would finish their tasks at exactly the same time. There are, of course, many parallel programs in which this regularity will not
Figura 9: Iteration space for the program in Figure 8.

```c
__global__ void depSumKernel(float* S, int side) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < side) {
        for (int j = 1; j < side; j++) {
            int ij = i * side + j;
            S[ij] = S[ij] - S[ij-1];
        }
    }
}
```

Figura 10: The CUDA program that is equivalent to the C program in Figure 5

occurr. Let’s take a look into the program in Figure 11, and its corresponding iteration space, in Figure 12.

A trivial, mechanical conversion of the program in Figure 11 into C for CUDA produces the program in Figure 13. This program has a wider iteration space than the original algorithm. This new iteration space is shown in Figure 14. The gray area represents the iterations where useful work is performed by the kernel. We use a conditional statement to
void digSum(float** s, unsigned int side) {
    int i, j;
    for (i = 1; i < side; i++) {
        for (j = 1; j < side; j++) {
            s[i][j] = s[i][j] - s[i-1][j-1];
        }
    }
}

Figura 11: C program illustrating temporal dependencies chains with different sizes.

Figura 12: Iteration space for the program in Figure 11.

avoid work performed outside the valid iteration space. This conditional has a very special semantics in the context of a SIMD hardware: when a processing element has no work to do, then it sleeps, until all the threads synchronize again, at the implicit barrier at the end of the conditional.

**Lowering the level** A program written in a high-level language is normally translated to binary code in order to be executed. With CUDA it is no different. There exist compilers
```c
__global__ void digSumKernel(float* s, int side) {
    int tx = blockIdx.x*blockDim.x + threadIdx.x;
    for (int it=1; it<side; it++) {
        int i = it;
        int j = tx - (side-1) + it;
        if (j >= 1 && j < side) {
            int ij = j + i * side;
            int ijp = j - 1 + (i - 1) * side;
            s[ij] = s[ij] - s[ijp];
        }
    }
}
```

Figure 13: The CUDA program that is equivalent to the C program in Figure 5

![Iteration space for the program in Figure 13.](image)

Figure 14: Iteration space for the program in Figure 13.

that translate CUDA to Direct3D vector instructions, for instance. We will be looking into an instruction set called Parallel Tread Execution, or PTX for short. The PTX version of the program in Figure 4 is given in the Figure 15. PTX is an assembly language; as such, it manipulate simple data types, such as integers (signed and unsigned, 8, 16, 32 and 64 bits) and floats (16, 32 and 64 bits). Because PTX supports the development of general purpose applications, it contains arithmetic (add, sub, mul, etc), logical (eq, leq, etc) and control flow (bra, ret, call, etc) instructions, in addition to usual memory access operations (load, store). Given the very nature of graphics applications, PTX contains many special instructions, mostly to represent transcendental functions such as square roots, sines, cosines and logarithms. Finally, PTX contains instructions that deal with parallel execution, such
as barrier synchronization, atomic arithmetic and logical operations and atomic memory accesses to shared memory.
Question 1.21 *The PTX instruction set contains some special variables. Given what we have seen so far, can you guess which variables would these be?*

**What is next?** There are two main sources of optimization in CUDA programming: the memory and the control flow. The second part of our course will describe memory optimizations in more detail, and the last part will talk about divergences, a control flow phenomenon that is typical in SIMD hardware. Stay tuned!
2 Memory optimizations

Going to the archives If you remember Computer Organization 101, then you know that the memory where the CPU gets its instructions and data is organized in a hierarchy. On the top of this hierarchy we have registers. Then we have caches (L1, L2, etc), RAM and hard-disks. Of course, this pyramid can have some more – or less – elements here and there, but you get the message. Just like the CPU, the GPU also organizes memory in a hierarchy. However, in the case of a GPU, the demand on the memory is much more intense.

Question 2.1 The GeForce 8800 process 32 pixels per clock. Each pixel contains a color (3 bytes) and a depth (4 bytes), which are read and written. On the average 16 extra bytes of information are read for each pixel. How many bytes are processed per clock?

Different GPU models use different memory layouts, but a fairly typical organization consists in separating memory into the following parts:

Registers the registers are fast, yet few. They are private to each thread.

Shared threads in the same block have access to a shared memory, which works like a cache.

Local each thread has access to a local memory space, in addition to its registers. Notice that the word “local” does not imply faster access: the local memory is off-chip, as as slow as the global memory.

Global the global memory is available to every thread in every block and every grid.

Shared, local and global memory, plus registers are located inside the GPU hardware. In addition to these four different storage locations, CUDA programs must also deal with a different, and rather alien type of memory: the DRAM that the CPU host uses to communicate with the GPU device. The DRAM is not really part of the GPU – it is more like a external driver. Thus, reading or writing into this memory is what we could call really slow. Some of the important players in the memory hierarchy are depicted in Figure 16.

There are a couple guidelines and tradeoffs that one should bear on mind when writing GPU programs.

1. Registers are the fastest storage location; however, they come in a very limited number. The GPU provides a fixed amount of registers to the thread block, say 1,024. The more registers the threads use, the less threads a block contains.

2. Having to go to the global or local memory is very slow. Ideally threads should be able to share as much data as possible in the shared memory. Remember: the shared memory is on-chip; therefore, it is much faster than the local and global memories, which are off-chip.
3. The inter-device communication, i.e., between the CPU and the GPU, is the most expensive. This channel is orders of magnitude slower than going to the shared memory, for instance, and should be minimized as much as possible. That is the main reason why GPUs are not suitable to interactive applications.

We will be talking in more detail about some of these considerations in the next sections.

**The interplanetary trip** Let’s compare the time that threads take to read data from the DRAM with the time to access this data through registers. We could say that the latter is the equivalent to cross the street to go to the bakery, whereas the former would correspond to a trip to the moon – walking. That is, to use the GPU, the CPU must transfer data to it, what is done through the Peripheral Component Interconnect (PCI) bus. This transfer is very slow, compared with accessing data on-chip.

The transfer of data between CPU and GPU is illustrated by the program in Figure 17. The functions `cudaMalloc`, `cudaFree` and `cudaMemcpy` are part of the CUDA programming library. The first function allocates data in the GPU memory space, while the second frees this storage space. Finally, `cudaMemcpy` copies data from CPU to GPU, or vice-versa, depending on its last argument.
void Mul(const float* A, const float* B, int width, float* C) {
    int size = width * width * sizeof(float);

    // Load A and B to the device
    float* Ad;
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd;
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd;
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid(wB / dimBlock.x, hA / dimBlock.y);

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, width, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}

Figura 17: The CPU code that calls a matrix multiplication kernel.

There are some aspects that we must consider, when writing CUDA programs, regarding the CPU/GPU communication. Firstly, it is only worthwhile sending work to the GPU if this work has high complexity. For instance, the program in Figure 5 is adding two matrices, cell by cell. The corresponding CUDA program, in Figure 7 assigns a constant, i.e, $O(1)$, amount of work to each processing element – and the constant, in this particular case, is pretty small. Thus, just the trouble of sending the matrices to the GPU would already eclipse any gain that we could obtain through parallel execution. In this case, the ratio of data transfer over operations performed is not good. In order to send back and forth the matrices $A$, $B$ and $S$, we would need $3N^2$ transfers. The kernel performs only $N^2$ additions. Hence, a ratio of 1:3. The story would be rather different were we talking about matrix multiplication. This algorithm
transfers the same amount of data than matrix addition, but performs $N^3$ additions and
multiplications, so the ratio of transfer operations over computation is $N^3/3N^2 = O(N)$.

**Question 2.2** Consider the programs in Figures 5, 8 and 11. Which programs are worth
parallelizing in the GPU, according to the previous discussion?

Secondly, once we send data to the GPU, it is important to keep it there as much as
possible. Data on the DRAM does not vanish away once the kernel that is using it terminates.
So, programs that use multiple kernels can avoid the interplanetary trip by simply invoking
kernels on data already sent to the GPU. Sometimes this approach is profitable even if the
work could be performed more efficiently in the CPU. A small kernel running on a sequential
problem setting could be faster than having to go for the expensive PCI bus.

**Question 2.3** Can you think about a situation like this, where you would have to leave a
kernel, do some computation on the CPU, and then call another kernel?

**The ballerina’s waltz** The heavy overhead of transferring data between CPU and GPU
can be mitigated if one can overlap the time spent on data transference and the time spent
on computation. CUDA gives the programmer a way of doing it, via the asynchronous
transmission of data. Asynchronicity can be obtained by transferring data from CPU to GPU
via the `cudaMemcpyAsync` function, instead of `cudaMemcpy`. The program in Figure 18, taken
from Nvidia’s best practices guide [4] compares the syntax for synchronous and asynchronous
data transference.

The synchronous program sends $nStreams$ streams of data from the CPU to the GPU,
and invokes a kernel to take care of each data stream. The `cudaMemcpyAsync` function is
non-blocking, that is, it gives control back to the CPU as soon as it is called, before the data
transfer is complete, contrary to cudaMemcpy. Figure 19 illustrates the overlapping of data that we obtain by using asynchronous data communication.

![Data transfer and Kernel execution diagram]

Figure 19: Synchronous versus asynchronous data transfer.

The Silk Road  During four centuries, the so called Silk Road was the main link between imperial China and growing Europe. “Road” would be, perhaps, an over-approximation, as it was more like a direction that travelers knew, and where they were certain to find company for the long trip, and lodging for the cold nights. This route was very long, and it would take months for goodies to travel from one end to the other. In the GPU world, the silk road is the path that data must journey between global memory and the thread local registers. This route is much shorter than the interplanetary voyage that we have discussed in the previous sections; however, it is still a long and tiring trip, which should be avoided whenever possible.

A good strategy to avoid going all the way along the silk road is to keep data in shared memory as much as possible. Reading or writing data into the shared memory is about 100 times faster than using the global memory. We will use the matrix multiplication example by Ryoo et al. [33] to illustrate this optimization. This kernel is the result of parallelizing the program in Figure 20.

**Question 2.4** On a PRAM model what is the best complexity that we can obtain to the matrix multiplication problem?

Writing the fastest solution to the matrix multiplication problem, in terms of asymptotic complexity, is a bit difficult, as you can quite well imagine. Therefore, we will settle for a solution that is $O(N)$ on a PRAM model, where $N$ is the width of the widest matrix. This parallel algorithm is given in Figure 21. Each thread is responsible for producing one single element in the final matrix $A$, i.e, $A_{ij}$. To produce this element, the thread must perform the dot product of the $i$-th line of matrix $B$, and the $j$-th column of matrix $C$.

**Question 2.5** Given Width = 10, how many access to the global memory the program in Figure 21 can perform in the worst case?
// Computes the matrix product using line matrices:
void mulMatCPU(float* B, float* C, float* A, unsigned int Width) {
    for (unsigned int i = 0; i < Width; ++i) {
        for (unsigned int j = 0; j < Width; ++j) {
            A[i * Width + j] = 0.0;
            for (unsigned int k = 0; k < Width; ++k) {
                A[i * Width + j] += B[i * Width + k] * C[k * Width + j];
            }
        }
    }
}

Figura 20: C program that performs the matrix product $A = B \times C$, using linearized matrices.

__global__ void matMul1(float* B, float* C, float* A, int Width) {
    float Pvalue = 0.0;
    for (int k = 0; k < Width; ++k) {
        Pvalue += B[threadIdx.y * Width + k] * C[k * Width + threadIdx.x];
    }
    A[threadIdx.x + threadIdx.y * Width] = Pvalue;
}

Figura 21: Kernel that performs the matrix product $A = B \times C$ equivalent to the program in Figure 20.

A good way to measure the performance of an application is to count how many floating-point operations it performs per second (FLOPS) [33]. The PTX code produced for the innermost loop in the program from Figure 21 is given in Figure 22. By looking at the PTX program, we know that about 1/8 of the instructions inside the loop are floating-point operations. To get this number we count the total number of instructions (16), and divide by the number of floating point operations (2, in the single multiply-add instruction). If we consider, for instance, the GTX 8800 GPU, then we have a hardware able to perform 172.8 billion floating-point operations per second. This gives us a raw throughput of 21.6 GFLOPS. Measurements would yield a much lower result: 10.58 GFLOPS [33]. The bottleneck, in this case, is memory.

The solution in Figure 21 is somehow naive, for the threads do not share data, even though they use a lot of common information. One-fourth of the instructions inside the main loop perform memory accesses to off-chip memory. In order to fully utilize the GPU, without stalls, we would need a bandwidth of 173GB/s (128 threads $\times 1/4 \times 4$ Bytes $\times$...
mov.f32 %f1, 0f00000000; // 0
mov.s32 %r10, %r5;

$Lt_0_1282:
//<loop> Loop body line 9, nesting depth: 1, estimated iterations: unknown
.loc 16 13 0
cvt.u64.u32 %rd3, %r7;
mul.lo.u64 %rd4, %rd3, 4;
.loc 16 9 0
ld.param.u64 %rd2, [__cudaparm__Z7matMul1PfS_S_i_B];
.loc 16 13 0
add.u64 %rd5, %rd2, %rd4;
ld.global.f32 %f2, [%rd5+0];
cvt.u64.u32 %rd6, %r9;
mul.lo.u64 %rd7, %rd6, 4;
.loc 16 9 0
ld.param.u64 %rd1, [__cudaparm__Z7matMul1PfS_S_i_C];
.loc 16 13 0
add.u64 %rd8, %rd1, %rd7;
ld.global.f32 %f3, [%rd8+0];
add.f32 %f1, %f2, %f3, %f1;
.loc 16 12 0
add.u32 %r7, %r7, 1;
.loc 16 9 0
ld.param.s32 %r3, [__cudaparm__Z7matMul1PfS_S_i_Width];
.loc 16 12 0
add.u32 %r9, %r3, %r9;
setp.ne.s32 %p2, %r7, %r8;
@%p2 bra $Lt_0_1282;
bra.uni $Lt_0_770;

Figura 22: The PTX version of the program in Figure 21.

1.35GHz), which is way larger than the featured bandwidth of 86.4GB/s. In the next section we will see how to improve this implementation.

The Tiles of Sherahzade A very popular technique to improve locality – hence reducing cache misses – is tiling [22]. The data to be manipulated is divided into blocks, and locations in the same block, assumed to be stored closely together, are used all at once. Tiling is particularly useful in our example, for we can partition the multiplication of two large matrices into the multiplication of several small matrices with a minimum of interference. The tiled program is given in Figure 23.

Question 2.6 Take a look into the program in Figure 23. How many thread blocks does it need to work correctly?
```c
__global__ void matMul2(float* B, float* C, float* A, int Width) {
__shared__ float Bs[TILE_WIDTH][TILE_WIDTH];
__shared__ float Cs[TILE_WIDTH][TILE_WIDTH];

int tx = threadIdx.x;
int ty = threadIdx.y;

// Identify the row and column of the A element to work on
int Row = blockIdx.x * TILE_WIDTH + tx;
int Col = blockIdx.y * TILE_WIDTH + ty;

float Pvalue = 0;
// Loop over the B and C tiles required to compute the A element
for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of B and C tiles into shared memory
    Bs[tx][ty] = B[Row*Width + (m*TILE_WIDTH + ty)];
    Cs[tx][ty] = C[Col + (m*TILE_WIDTH + tx)*Width];
    __syncthreads();

#pragma unroll 1
for (int k = 0; k < TILE_WIDTH; ++k)
    Pvalue += Bs[tx][k] * Cs[k][ty];
    __syncthreads();
}
A[Row*Width+Col] = Pvalue;
}
```

Figura 23: Tiled matrix multiplication kernel.

**Question 2.7** Still considering the program in Figure 23, we see that the tile size is determined by the variable TILE_WIDTH, which is customizable. What are the tradeoffs that one must consider when choosing the right tile size?

If your head started spilling some smoke after Figure 23, no worries: this program is not exactly easy to understand. Normally we would compute one cell of the product matrix by going over a full line of matrix A, and a full column of matrix B. However, in the tiled algorithm we are doing it in parts. We bring a tile from matrix A to the shared memory, and do the same with a tile from matrix B. We then operate on these tiles, yielding a partial product for a number of cells. Each of these products is not the final value that will go to matrix A. To obtain this value, we need to go over all the tiles containing a full line, and a full column. Figure 24 illustrates the technique. We are assuming two $4 \times 4$ matrices, and $2 \times 2$ tiles. The figure shows the two iterations necessary to produce the left-upper tile of
matrix $A$.

**Question 2.8** What is the floating-point instruction throughput in the inner iteration of the kernel in Figure 23?

**Question 2.9** The program in Figure 21 reads each cell of either matrix $B$ or $C$ a number of times equal to $\text{Width}$. What about the tiled version in Figure 23?

**Question 2.10** What is the memory bandwidth of the tiled program in Figure 23, considering that the bandwidth of the original program in Figure 21 is 173GB/s?

**Question 2.11** Before moving on, is there anything we could do to improve the throughput of the program in Figure 23?

**Back to the Future II** In the nerd cult movie *Back to the Future II*, Tip, the bad guy, ends up visiting the future, accidentally, and finds a catalogue with the result of the major sport events in the US. He returns to the present time, and uses this guide to win on the lottery, time after time; hence, becoming the “luckiest” man in the world. Like Tip, we can use the gift of prophecy to produce better CUDA code. Of course, it is very hard to know what will happen in the future... unless we are the ones who “make” the future. For instance, if we know which data is going to be used next, then we can *prefetch* it from memory before
__global__ void matMulPref(float* B, float* C, float* A, int Width) {
  __shared__ float Bs[TILE_WIDTH][TILE_WIDTH];
  __shared__ float Cs[TILE_WIDTH][TILE_WIDTH];
  int tx = threadIdx.x;
  int ty = threadIdx.y;
  // Identify the row and column of the A element to work on
  int Row = blockIdx.x * TILE_WIDTH + tx;
  int Col = blockIdx.y * TILE_WIDTH + ty;
  float Pvalue = 0;
  // Data pre-fetching:
  float tmpB = B[Row * Width + ty];
  float tmpC = C[Col + tx * Width];
  int limit = Width / TILE_WIDTH;
  // Loop over the B and C tiles required to compute the A element
  int m = 0;
  while(m < limit) {
    // Collaborative loading of B and C tiles into shared memory
    Bs[tx][ty] = tmpB;
    Cs[tx][ty] = tmpC;
    __syncthreads();
    m++;
    if (m < limit) {
      tmpB = B[Row*Width + (m*TILE_WIDTH + ty)];
      tmpC = C[Col + (m*TILE_WIDTH + tx)*Width];
    }
  #pragma unroll 1
  for (int k = 0; k < TILE_WIDTH; ++k)
    Pvalue += Bs[tx][k] * Cs[k][ty];
  __syncthreads();
  }
  A[Row*Width+Col] = Pvalue;
}

Figura 25: Tiled matrix multiplication kernel with prefetching of data.

it is really needed, in this way overlapping on the hardware pipeline cycles spent on memory access and computation.

Some instruction sets would give the compiler a “prefetch” instruction that would point to the hardware which data was the next to be used. Cuda does not offer anything similar; however, the developer can try to overlap memory and computation cycles at the software level. Figure 25 shows how to perform this kind of optimization on the program in Figure 23. In the new program, data from array C is fetched before the iteration where it is needed, in such a way that this memory access overlaps with the computation performed by the innermost for loop.
The Oscar Night  During the Oscar warding night Hollywood receives many celebrities, which enter the Theater Hall on a glamorous red carpet. Unrolling this carpet requires a good deal of effort, yet, it improves performance. You do not know how? Well, if we want to have better throughput, the basic strategy is to minimize the amount of non-floating point operations inside the loop.

Question 2.12 **Looking back at the program in Figure 23, which operations inside the innermost loop are not floating-point instructions?**

Every loop contains a branch that normally applies some condition on an induction variable, which, by the way, eats off one register. Loop unrolling is a classic compiler optimization that removes these branches, and may end up lowering the register pressure on the program, if it succeeds in eliminating the induction variable. We continue following the example of Ryoo et al. [33], and show, in Figure 26 the result of doing loop unrolling on the program in Figure 23. In this new version of our matrix multiplication kernel we assume that the tile width is set to be 16. Going against all the good coding standards, we hardwire this value directly into the code, just to make it explicit that this code works for this tile size only.

Question 2.13 **Try compiling the program in Figure 26 and take a look into its PTX code. Which percentage of instructions do floating-point work? What is the potential throughput of the new program?**

Given that we have unrolled completely the loop, we no longer need a register to store variable k in Figure 23. Thus, our loop unrolling optimization has been able to reduce the register pressure in this kernel. This is a good thing, as we will soon see, when we start talking about register allocation.

The tale of the Arab merchant  Salameh al Korezmi was a prosperous merchant who worked along the silk road, bringing silk from China to Italy. Salameh had a camel, which could carry 16 loads of silk, which he got from 16 different Chinese producers, all living in the same province of Mu Peng. Salameh lived quite happy then, until one unfortunate day, when he had discovered that one of his producers had moved away from Mu Peng, to the province of Li Peng. The sad part of this story is that Mu Peng was at war with Li Peng – possibly because nobody could agree on which town produced the best silk. So, Salameh could not really leave Mu Peng and go to Li Peng, for he would be called a spy. He had to go all the away from Italy to Mu Peng, get 15 loads of silk, bring them back to Europe, and then repeat the same trip towards Li Peng, to get the other load. Because of the war, Salameh’s access to the silk was not *coalesced*.

In CUDA we may face the same problem as Salameh’s. Different CUDA execution models group data in different ways; however, we may assume that the global memory is divided into *segments* that fit 16 data-words. Let’s say that each segment is at war with the others, and a transaction cannot take data from two different segments in the same trip – or one may think that the transaction is doing some spying. Therefore, in order to avoid multiple trips between shared and global memory, we better make sure that our loads of silk are in the same town; or that the data which we need are on the same segment.
__global__ void matMul3(float* Md, float* Nd, float* Pd, int Width) {
__shared__ float Mds[16][16];
__shared__ float Nds[16][16];

int tx = threadIdx.x, ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
int Row = blockIdx.x * 16 + tx, Col = blockIdx.y * 16 + ty;

float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
for (int m = 0; m < Width/16; ++m) {
  // Coolaborative loading of Md and Nd tiles into shared memory
  Mds[tx][ty] = Md[Row * Width + (m * 16 + ty)];
  Nds[tx][ty] = Nd[Col + (m * 16 + tx) * Width];
  __syncthreads();

  Pvalue += Mds[tx][ 0] * Nds[ 0][ty];
  Pvalue += Mds[tx][ 1] * Nds[ 1][ty];
  Pvalue += Mds[tx][ 2] * Nds[ 2][ty];
  Pvalue += Mds[tx][ 3] * Nds[ 3][ty];
  Pvalue += Mds[tx][ 4] * Nds[ 4][ty];
  Pvalue += Mds[tx][ 5] * Nds[ 5][ty];
  Pvalue += Mds[tx][ 6] * Nds[ 6][ty];
  Pvalue += Mds[tx][ 7] * Nds[ 7][ty];
  Pvalue += Mds[tx][ 8] * Nds[ 8][ty];
  Pvalue += Mds[tx][ 9] * Nds[ 9][ty];
  Pvalue += Mds[tx][10] * Nds[10][ty];
  Pvalue += Mds[tx][11] * Nds[11][ty];
  Pvalue += Mds[tx][12] * Nds[12][ty];
  Pvalue += Mds[tx][13] * Nds[13][ty];
  Pvalue += Mds[tx][14] * Nds[14][ty];
  Pvalue += Mds[tx][15] * Nds[15][ty];

  __syncthreads();
}
Pd[Row*Width+Col] = Pvalue;
}

Figura 26: Matrix multiplication kernel after loop unrolling.

**Question 2.14** How to ensure that all the threads in a half-warp (16 threads) access the same segment of data?

Question 2.14 is not that difficult. Basically, we perform the following two checks:
__global__ void offsetCopy(float *odata, float* idata, int offset) {
    int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
    odata[xid] = idata[xid];
}

Figura 27: A simple copy kernel that demonstrates coalesced versus uncoalesced memory access.

1. take thread \( t \) whose id is \( 16 \times n \).
2. find out the segment \( s \) that thread \( t \) is accessing.
3. for every thread \( t + i, 1 \leq i \leq 15 \), see if \( t + i \) is using segment \( s \).

**Question 2.15** Consider the program in Figure 26. Is this program performing coalesced memory accesses?

**Question 2.16** What would you expect to happen if the half-warp access data that is in a successive sequence, yet not aligned on the same segment?

**Question 2.17** Consider the program in Figure 27. Try varying \( \text{offset} \) from 0 to 31, and plot the time results you obtain. Use large vectors of data that are multiple of 16.

**Register to remember** In both, the GPU and the CPU, registers are the fastest storage location. However, whereas the CPU has very few registers, the GPU provides a bountiful lot. For instance, the x86 normally gives applications eight general purpose registers, and some of these are already taken for particular purposes, like points at the stack of activation records. On the other hand, the GeForce 8800 GPU has 16 streaming multiprocessors, each one having 8,192 registers. The caveat is that these registers must be partitioned among up to 768 threads. Thus, if an application uses all the available threads, then it will end up giving only 10 registers to each thread.

Therefore, there exists a tension between the maximum number of registers that we allocate to each thread, and the maximum number of threads that we can run simultaneously. For instance, the program in Figure 21 uses 10 registers. Hence, it can run up to 768, for \( 10 \times 768 < 8,192 \). This means that we can schedule three thread blocks to run together, each one having 256 threads, which is the maximum allowed by the hardware. However, if the register pressure in this program were one unit larger, then we would have \( 256 \times 11 \times 3 = 8,488 > 8,192 \). In this case, each streaming multiprocessor would execute only two thread blocks simultaneously.

**Question 2.18** Ok, register pressure is a key concept, but, how do we compute it for a given program?
foo (int p) {
  x = p + 1
  y = x + p
  return y
}

Figura 28: Register pressure in a linear code sequence.

The register pressure in a program depends on the number of variables that are simultaneously alive at a given program point. We say that a variable is alive at some program point if it holds a value that can be used in the future from that program point. Figure 28 illustrates these notions. We have a function that declares a parameter and two local variables. The parameter and variables need a location, which ideally would be registers. The live ranges of these entities are shown as black line segments. As we can see, the live range of $y$ does not overlap with neither $p$’s nor $x$’s; thus, $y$ can share a register with any of these variables. In the Figure we have allocated $y$ together with $x$.

The register pressure in the example of Figure 28 is two – the maximum number of overlapping live ranges. This example is an easy case, because it does not contain control flow. Finding the register pressure in general programs is an NP-complete problem, so, compilers resort to heuristics, like graph coloring. Appel and Palsberg give a throughout explanation about a graph coloring based method to determine the register pressure of programs [5].
3 Divergence Analyses and Optimizations

Most of the optimizations that we have seen in Section 2 are classic techniques, which can be used to improve sequential C programs, just as well as high-tech GPU programs. Tiling, prefetching and register allocation, for instance, are “tale as old as time” [1]. However, there exists a class of optimizations that apply only to SIMD execution models, like the model that supports CUDA warps. This peculiarity is due to a phenomenon that we call divergent execution. In this section we will talk about divergences, and will try to answer questions like:

- what are divergences;
- how to detect divergences during program execution;
- how to manually optimize code to mitigate the effects of divergences;
- how to predict the locations where divergences can happen;
- which automatic optimizations can we apply on divergent code.

The Shangri La Monks  Lost Horizon is a 1933 novel by James Hilton, which became famous for describing the Tibetan town of Shangri La. Shangri La was a utopia: people would be always happy, sharing everything, while despising money and properties. Hiton never quite had the time to describe the monks who lived in the Shangri La temple, so, here I will do him this favor, for the sake of obtaining a good metaphor 3. The Shangri La monks live in such a perfect union, that they always do the same things together. When they eat, they all eat together. When they dance, they all dance together. When they pray, they all pray together. To avoid dissension, if two groups of monks disagree about the next thing to do, then one of the groups take a nap, while the other performs its duty. Once the first group finishes, they wake up their sleeping brothers, and take their place in the holy mattresses. After the second group of monks do their stuff, they call the first group, all of them take hands, and go to do things together again. Life goes back to its ordinary course, and harmony rules again in Shangri La. The threads inside a CUDA warp behave exactly like the monks...

GPUs are highly parallel; however, due to its restrictive programming model, not every application can benefit from all the processing power that they provide. In these processors, threads are organized in groups that execute in lock-step. Such groups are called warps in the Nvidia jargon 1, or wavefronts in ATI’s 2. To better understand the rules that govern the behavior of threads in the same warp, we can imagine that each warp might use a number of processing units, but has only one instruction fetcher. Let’s illustrate these concepts with a real example: the GeForce 8800 GPU is able to run 16 warps at the same time; each warp consists of 32 threads, and uses 8 processing units. Thus, each warp might perform 32 instances of the same instruction in four cycles of the hardware pipeline. In

3Grace Hoper would always say: “asking for forgiveness is easier than asking for permission.”
very regular applications, such as scalar vector multiplication, we have the same operation been independently performed on different chunks of data. These applications fare very well on GPUs. However, not every application is so regular, and divergences may happen. Informally, we say that a program is divergent if, during its execution, threads inside the same warp follow different paths after processing the same branch.

Divergences occur due to conditional branches. The branching condition might be true to some warp threads, and false to others. Given that each warp has access to only one instruction at each time, in face of a divergence, some threads will have to wait, idly, while their sister threads execute. Therefore, divergences may be a major source of performance degradation in GPU applications. As an example, Baghsorkhi et al [7] have analytically found that approximately one third of the execution time of the simple prefix scan benchmark [19], included in the CUDA software development kit (SDK), is lost due to divergent program flows. Optimizing an application to avoid divergences is problematic for a number of reasons. First, some parallel algorithms are inherently divergent; thus, threads will naturally disagree on the outcome of branches. Second, finding the program points that account for the most serious divergences burdens the application developer with a difficult and tedious task, which requires a deep understanding of code that might be large and convoluted.

**Question 3.1** Let’s see if you understood it: why are divergences a problem in GPUs, but are not even a concern in other parallel execution environments, such as those based on the SPMD paradigm?

**(Bi-)Tonic Vodka** Sorting in parallel is not the same as multiplying matrices. There is a lot of coordination that must be performed. A good approach to this problem is based on the notion of sorting networks.

**Question 3.2** We are drifting a little bit here, but tell me: what are sorting networks, and what do they have to do with parallelization?

Probably the most well-known sorting network based algorithm is bitonic sort [8]. This algorithm suffers due to the $O(n \ln^2 n)$ complexity when restricted to a sequential hardware; nevertheless, the ability to perform $n/2$ comparisons, where $n$ is the array size, in parallel, makes this algorithm attractive to the GPU execution environment. In order to illustrate the concepts introduced in this section we will show how they apply to a popular implementation of the bitonic parallel sorting algorithm ⁴. Bitonic sorting is often used as a component of other sorting algorithms; for instance, it is used in the parallel quicksort implementation of Cederman and Tsigas to sort small arrays [9]. The CUDA implementation, as taken from the SDK, is given in Figure 29.

The CUDA implementation in Figure 29 is hopelessly tied to our single instruction multiple data execution model. That is, we have multiple functional units, but only one instruction fetcher. Thus, invariably the condition $(tid \& k) == 0$, where tid is the thread identifier, will be true to some threads, and false to others. In this case, we will have a divergence

³http://www.cs.chalmers.se/~dcs/gpuqsortdcs.html
__global__ static void bitonicSort(int * values) {
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;
    shared[tid] = values[tid];
    __syncthreads();
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        for (unsigned int j = k / 2; j>0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0) {
                    if (shared[tid] > shared[ixj]) {
                        swap(shared[tid], shared[ixj]);
                    }
                } else {
                    if (shared[tid] < shared[ixj]) {
                        swap(shared[tid], shared[ixj]);
                    }
                }
            }
            __syncthreads();
        }
    }
    values[tid] = shared[tid];
}

Figura 29: Nvidia’s SDK implementation of Bitonic Sort.

forcing threads to wait while others perform work. In order to see how much divergences degrade performance, let’s leave the high level code from Figure 29 aside, to focus, instead, on the simplified PTX program in Figure 30.

**Question 3.3** Figure 30 uses a notation that we have not yet explained. We call it the Control Flow Graph (CFG). Answer the following questions about the CFG:

1. What are the boxes?
2. What are the boldface numbers in front of the boxes?
3. What are the arrows?
4. Can you guess which part of Figure 29 is represented in Figure 30?
5. Can you think about an algorithm that converts a linear sequence of PTX instructions into a control flow graph?
Figura 30: The (simplified) PTX representation of part of the inner loop of the bitonic kernel in Figure 29.

Figure 31 shows a warp trace in a setting where we have four functional units, and a single warp with four threads, that we have called \( t_0, t_1, t_2 \) and \( t_3 \). We assume that values, the input array, is \( \{4, 3, 2, 1\} \), and that the id of thread \( t_n \) is \( n, 0 \leq n \leq 3 \). When \( k = 2 \) and \( j = 1 \), the input array causes two divergences. The first split happens at cycle \( i = 3 \), due to the branch \( \text{bra } \%p1, \text{L2} \), and it separates \( t_0 \) and \( t_2 \) from \( t_1 \) and \( t_3 \). This divergence happens because the condition \( \%ixj > \%tid \) is true only for \( t_0 \) and \( t_2 \). The second split happens at cycle \( i = 6 \), due to the branch \( \text{bra } \%p2, \text{L3} \), and it separates threads \( t_0 \) and \( t_2 \).

Question 3.4 Take a new look into Figure 31 and tell me: what is the cost of the divergences in this particular example, in terms of latency and throughput?
Figura 31: A snapshot of the warp trace over the program in Figure 30, showing the first iteration of that code, assuming $|w| = 4$, and $v = \{4, 3, 2, 1\}$.

Playing Sherlock  Now that we know what divergences are all about, how can we find them out? I mean, how to find the branches that cause divergences? These are the program points that one should pay attention when trying to optimize the code to mitigate the problem of divergences. There are two basic approaches to investigate divergences: profiling and static analysis. Profiling is a dynamic technique: we run the program in a monitored environment, in such a way to be able to record in which branches divergences have happened. Static analysis is just the contrary: we take a good look into the code, and try to prove that a branch will always diverge, or will never do it. Generally proving the impossibility is easier than proving the occurrence, for a branch may diverge or not, depending on the program input. Thus, we normally stick to showing that a branch will never cause a divergence. We will talk about profiling first, and leave static analysis for later.

Profilers are old allies of compiler designers. Since the debut of the highly popular and influential gprof tool, introduced by Graham et al. [18], many profiling tools and techniques have been described. Profiling fits three main purposes: first, it helps application developers to find performance bottlenecks in their programs. Second, it guides compilers into performing more aggressive code optimizations, as nicely described by Chang et al. [10]. Finally, profilers facilitate debugging. A classic example in this last category is the Valgrind memory profiler [28].

Profiling tools have been initially used to measure the dynamic behavior of sequential applications [18, 24, 28, 36]. Subsequent works have extended profiling into the realm of parallel computers with great success [15, 20, 25, 37, 41]. Parallel profilers, such as the
recent works of Tallent et al. [37] or Eyerman and Eeckhout [15] generally focus on systems where threads are able to execute independently of each other. Profiling applications that fit on the GPU execution model is, thus, still a challenge.

Question 3.5 Ok: obviously we are going to profile GPUs to detect divergences. Yet, there are a lot of numbers that we could try to get via profiling. Can you name a few?

There are at least two profilers that can capture the divergent behavior of Cuda code. On the industrial side, Nvidia has released the CUDA Visual Profiler, or CVP, as this application is normally known. CVP lets the CUDA developer to probe several aspects of the kernel behavior, including the existence of divergent threads in the parallel program. However, CVP does not point in which program points the divergences have happened. In order to obtain this more precise information, one can use the divergence map introduced by Coutinho et al. [12].

A democratic profiler Profiling Cuda programs to detect divergences is a bit different than profiling sequential programs. There are many threads running in parallel, and they execute in lock-step. A profiler must be aware of all these threads. In particular, a profiler must be able to see that after going over a branch some threads chose a path, whereas others did not. We do this kind of profiling via instrumentation.

Question 3.6 There are more than one way to implement profilers. For instance, we can use emulators, sampling or instrumentation. Can you explain each one of these techniques?

We store the results of profiling into a data-structure called the divergence map. This data-structure consists of two arrays of integers, that we call $\tau$ and $\delta$, such that $\tau[b]$ stores the number of warps that have visited basic block $b$, and $\delta[b]$ stores the number of divergences that took place at $b$. We insert the necessary instrumentation automatically in a three-phase process:

1. **Initialization**: when we reserve memory on the GPU address space to store our arrays, and initialize these arrays with zero’s.

2. **Measurement**: when we compute the number of visits and divergences, storing the results in our arrays.

3. **Reading**: when we copy to the CPU the data accumulated during the profiling of the kernel program, in the GPU.

The initialization and reading phases are trivial; thus, we will only describe the measurement phase.

In order to check the occurrence of divergences we can insert the code in Figure 32 at each branch. This figure shows the instrumentation of block $L_2$ of the example in Figure 30. We

Figura 32: The code inserted to instrument block $L_2$ in Figure 30. Above we have a higher level description of the instructions in block $L_2(\text{find writer})$. Code from the original program is marked in gray.

split each instrumented basic block $b$ into three new blocks: $b_{\text{up}}$, $b_{\text{bottom}}$ and $b_{\text{find writer}}$. The code that performs the instrumentation executes two tasks: (i) in blocks $b_{\text{up}}$ and $b_{\text{find writer}}$ we find a thread – the writer – to update the arrays $\delta$ e $\tau$. (ii) in block $b_{\text{bottom}}$ we detect and report the divergence.
We let the writer to be the thread with the lowest identifier among the active threads in the warp. In Figure 32 the variable \%laneid denotes the thread identifier inside the warp. In blocks \( b_{up} \) and \( b_{\text{find writer}} \) we loop through the live threads, looking for the one with lowest \%laneid.

**Question 3.7** Why we cannot simply choose as the writer the thread with \%laneid = 0?

Once we have a suitable writer, we perform the detection of divergences via voting. The PTX instruction \%p = vote.uni.pred, \%q sets \%p to true if all the threads in the warp find the same value for the predicate \( q \). Thus, we vote on the predicate that controls the outcome of a potentially divergent branch. If every warp thread agrees on the predicate, no divergence happens at that particular moment of the program execution; otherwise, a divergence takes place, and it is necessary to increment the \( \delta \) array. The instruction \( @iAmWriter \%\delta[L_2] = \) \( \text{atom.add} \%\delta[L_2] \), in the ninth program point in Figure 32, adds one to \( \delta[L_2] \) if, and only if, the predicate \( @iAmWriter \) is true.

**Question 3.8** What is the cost per branch, in terms of asymptotic complexity, of the code that performs the instrumentation? Explain your analysis in term of the number of warps in the kernel, and the number of threads per warp.

**Question 3.9** Going back to Question 3.7, is the worst case complexity different from the average case?

As you probably have guessed when answering Questions 3.8 and 3.9, instrumentation adds a huge burden on the profiled program. In terms of code size, the instrumented program tends to grow between 2 and 3 times. The overhead is even bigger in terms of time, multiplying the execution time of applications by a factor of up to 1,500x! Nevertheless, these numbers are similar to other instrumentation based profiling strategies [27]. More important: the instrumentation does not change the semantics of the program, as it does not use any of the program variables. Hence, by observing divergences we do not change the pattern of divergences in the program, what is exactly the information that we want to measure.

**Fast and furious** Let’s see how we can use profiling information to optimize the bitonic sort implementation from Figure 29. The implementation of quicksort of Cederman et al. [9] uses the traditional quicksort to partition a large array of numbers into small chunks of data, which are then sorted via the algorithm shown in Figure 29. The speedup numbers that we give in this section refer to the whole quicksort algorithm, even though we only change the bitonic sort kernel. That is, we deliver 6-10% performance speed up by changing less than 10-12 assembly instructions in a program containing 895 instructions!

Figure 33 shows some of the profiling results that we obtain for bitonic sort. This figure tells us that the suite of conditionals nested into the doubly nested loop suffers from a large number of divergences. Specifically, the condition \((\text{tid} \& k) == 0\) is traversed by warps 28
__global__ static void bitonicSort(int * values) {
    extern __shared__ int shared[];
    const unsigned int tid = threadIdx.x;
    shared[tid] = values[tid];
    __syncthreads();
    for (unsigned int k = 2; k <= NUM; k *= 2) {
        for (unsigned int j = k / 2; j > 0; j /= 2) {
            unsigned int ixj = tid ^ j;
            if (ixj > tid) {
                if ((tid & k) == 0) {
                    if (shared[tid] > shared[ixj]) {
                        swap(shared[tid], shared[ixj]);
                    }
                } else {
                    if (shared[tid] < shared[ixj]) {
                        swap(shared[tid], shared[ixj]);
                    }
                }
            }
        }
        __syncthreads();
    }
    values[tid] = shared[tid];
}

Figura 33: Divergence results obtained via profiling using the standard input for the Ceder- man et al. implementation of parallel quicksort [9].

million times, and one third of these visits diverge. The map also shows that divergences are common in both the sub-clauses of this branch, namely shared[tid] > shared[ixj] and shared[tid] < shared[ixj].

In order to improve the kernel, we start by noticing that the code trace formed by block L3 followed by block L5 is very similar to the trace L4 + L6. The only difference comes from the conditionals in program points 9 and 13. Armed with this observation, we bind the blocks L5 and L6 together, using the index manipulation trick that gives us the program in Figure 34 (a). Divergences might still happen; however, whereas the maximum divergent path in Figure 30 contains eight instructions, the worst divergence case in Figure 34 (b) contains only six instructions. This optimization gives a speed-up of 6.75% on a GTX 8800.

The code in Figure 34 still provides us with optimization opportunities. We can use the ternary selectors available in CUDA to merge basic blocks L3 and L4, thus removing the divergence at the end of block L2. The modified source is shown in Figure 35 (a), and the equivalent PTX code is given in Figure 35 (b). An instruction such as %a = sel %tid %ixj %p assigns %tid to %a if %p is not zero. It assigns %ixj to %a, otherwise. In the new program,
unsigned int a, b;
if (((tid & k) == 0){
    b = tid;
    a = ixj;
} else {
    b = ixj;
    a = tid;
}
if (sh[b] > sh[a]){
    swap(sh[b], sh[a]);
} else {
    b = ixj;
    a = tid;
}
if (sh[b] > sh[a]){
    swap(sh[b], sh[a]);
}

int p = (tid & k) == 0;
unsigned b = p?tid:ixj;
unsigned a = p?ixj:tid;
if (sh[b] > sh[a]) {
    swap(sh[b], sh[a]);
}

the worst case divergent path has only two instructions. This final optimization gives us a speed-up of 9.2%.

**Question 3.10** Look at the outermost if statement in Figure 33, which uses the condition \(ixj > tid\). Although we have not provided divergence data for this branch, do you think
you could optimize it, in case it were heavily divergent?

**Question 3.11** Let’s compare the programs in Figures 33 and 34. Which code do you think would run faster in a sequential machine, if we did not allow further compiler optimizations, i.e., we compile them with `gcc -O0`?

**Castor and Pollux** A profiler is very useful, as we have seen when optimizing bitonic sort. However, it has some drawbacks:

1. the profiling information is heavily input dependent. If we feed the profiled program with data that will not cause divergences, then we obtain a misleading histogram of divergences.

2. Instrumentation-based profiling may increase the execution time of the instrumented program over 1000x!

3. The profiler cannot guarantee that a divergence will never happen at a given branch.

The last item in this list is particularly troublesome: in order to implement optimizations automatically, the compiler must have some guarantees about the behavior of the code that is being modified. We can circumvent these deficiencies of the profiler using static analysis techniques.

Notice that static analyses have problems too. The main trouble, in this case, is that in order to be always correct, many static analyses end up being very conservative. That is, the static analysis tends to mark as divergent many more branches than the dynamic profiler. Nevertheless, these two techniques, profiling and static analysis, complement each other, often providing more information than the sum of the individual parts. The profiler, for instance, is useful to fine-tune the static analyzer. Static analysis, on its turn, helps reducing the amount of instrumentation that the profiler must insert into the program.

**Question 3.12** How can we use static analyses to reduce the amount of instrumentation that the profiler must insert in the program?

Given a Cuda program $P$, we are interested in determining a conservative, yet non-trivial, approximation of the set of divergent variables of $P$. A variable $v$ is divergent if there exist two threads in the same warp such that each thread sees $v$ with a different value at a given moment during program execution. In the next sections we will describe divergence analysis, a static analysis that finds the set of divergent variables. This analysis is similar to Aiken and Gay’s barrier inference analysis [2]; however, whereas barrier inference assumes a SPMD execution model and a high-level language, divergence analysis run on a SIMD assembly language. In order to explain how it works, we will need some concepts that exist exiled in the realm of compiler books. We introduce these expatriates in the next section.
Highlanders  The highlander was an outcast strolling on Earth, bearing the “heavy curse” of living forever – unless someone chop off his head. The curse contract had, in very small letters, that the highlander would have to kill the other of his kin, for, in the world where he duels, there must be only one. There exists a very famous compiler intermediate representation that abides by the highlander contract – the Static Single Information form (SSA) [13]. In this program representation we can have only one instance of each variable name. Normally it is easier to implement compiler analyses and optimizations on SSA-form programs; therefore, we will assume that all our PTX programs from now on are given in this representation.

\[
\begin{align*}
%b_1 &= \text{mov} \ %\text{tid} \\
%a_1 &= \text{mov} \ %\text{ixj} \\
%b_2 &= \text{mov} \ %\text{ixj} \\
%a_2 &= \text{mov} \ %\text{tid} \\
%a &= %a_1 \ %a_2 \\
%b &= %b_1 \ %b_2 \\
%t_2 &= \text{ld} \ %\text{shared}[\%b] \\
%t_3 &= \text{ld} \ %\text{shared}[\%a] \\
%p_3 &= \text{gt} \ %t_2 \ %t_3 \\
\text{bra} \ %p_3 \ L_7
\end{align*}
\]

\[
\begin{align*}
%b_1 &= \text{mov} \ %\text{tid} \\
%a_1 &= \text{mov} \ %\text{ixj} \\
%b_2 &= \text{mov} \ %\text{ixj} \\
%a_2 &= \text{mov} \ %\text{tid} \\
%a &= %a_1 \ %a_2 \\
%b &= %b_1 \ %b_2 \\
%t_2 &= \text{ld} \ %\text{shared}[\%b] \\
%t_3 &= \text{ld} \ %\text{shared}[\%a] \\
%p_3 &= \text{gt} \ %t_2 \ %t_3 \\
\text{bra} \ %p_3 \ L_7
\end{align*}
\]

\begin{figure}
\centering
\begin{tabular}{cc}
\hspace{0.5cm} L3 & L4 \\
\hspace{0.5cm} %b = \text{mov} \ %\text{tid} & %b = \text{mov} \ %\text{ixj} \\
\hspace{0.5cm} %a = \text{mov} \ %\text{ixj} & %a = \text{mov} \ %\text{tid} \\
\end{tabular}
\begin{tabular}{cc}
\hspace{0.5cm} L3 & L4 \\
\hspace{0.5cm} %b_1 = \text{mov} \ %\text{tid} & %b_2 = \text{mov} \ %\text{ixj} \\
\hspace{0.5cm} %a_1 = \text{mov} \ %\text{ixj} & %a_2 = \text{mov} \ %\text{tid} \\
\end{tabular}
\caption{The conversion to Static Single Assignment form (SSA). (a) Original program. (b) SSA-form program.}
\end{figure}

Figure 36 illustrates the conversion of programs into SSA form using part of the program in Figure 34. This conversion is done by creating new names to variables that have the same name, and by inserting \( \phi \)-functions to join the different names of a variable into a common name. For instance, the name \( a \) has two occurrences in Figure 36 (a); hence, the program is not in SSA-form. We sort this problem out by renaming these instances to \( a_1 \) and \( a_2 \). But now we have another problem: variable \( a \) was used at basic block \( L_{3/4} \); so, which should be the name used at that block, \( a_1 \) or \( a_2 \)? The answer is, neither one – and all of them. If the program flows from block \( L_3 \) into \( L_{3/4} \), then the correct name is \( a_1 \), otherwise, control came from block \( L_4 \), and the correct name to be used is \( a_2 \). We sort this out by inserting a \( \phi \)-function at the beginning of \( L_{3/4} \), e.g: \( a = \text{phi}(a_1, a_2) \). The \( \phi \)-function works like a multiplexer, assigning to \( a \) either \( a_1 \) or \( a_2 \), depending on the program flow.

\textbf{Question 3.13} \( \phi \)-functions are an abstract notation – they do not really exist in the assembly program, not least on Nvidia’s PTX. How are these instructions implemented in the real world?

In addition to the notion of SSA-form programs, we need also the concept of post-dominance. Let’s assume that each instruction in a PTX control flow graph is represented
by a label $l$. A label $l_p$ post-dominates a label $l$ if, and only if, every path from $l$ to the exit of the program goes across $l_p$. Furthermore, we say that $l_p$ is the immediate post-dominator of $l$ if $l_p \neq l$, and any other label that post-dominates $l$ also post-dominates $l_p$. Fung et al. [17] have shown that re-converging divergent threads at the immediate post-dominator of the divergent branch is nearly optimal with respect to maximizing hardware utilization. Although Fung's work has also discovered situations in which it is better to do this re-convergence past $l_p$, they are very rare. Thus, we assume that the immediate post-dominator $l_p$ of a divergent branch always contains an implicit synchronization barrier, which we will describe by a $\text{sync}$ instruction.

Finally, we must also define the notion of program path. Given an assembly program $P$, we say that a label $l$ goes to a label $l'$, what we indicate by $l \rightarrow l'$, if one of these three conditions hold: (i) $l' = l + 1$ and $l$ is not a jump, (ii) $l$ is a branch that may jump to $l'$, or (iii) $l$ is a jump that targets $l'$. We say that $l_1 \xrightarrow{*} l_n$ if, either $l_1 = l_n$, or $l_1 \rightarrow l_2$ and $l_2 \xrightarrow{*} l_n$. If $l$ is a label, and $l_p$ is its immediate post-dominator, then we define the influence region $\text{IR}(l)$ as the union of every path $l \xrightarrow{*} l_p$ that contains $l_p$ exactly once (and thus at the end).

We say that a variable $v$ from a program $P$ reaches a label $l$ if $P$ contains two labels $l_d$ and $l_u$ such that: (i) $v$ is defined at $P[l_d]$, (ii) $v$ is used at $P[l_u]$, (iii) $P$ contains a path $l_d \xrightarrow{*} l$, and (iv) $P$ contains a path $l \xrightarrow{*} l_u$.

**Question 3.14** Figure 37 is an artificial example that we will use throughout the next sections. Answer the following questions about this program:

1. Which instruction is the immediate post-dominator of the instruction $\text{branch } \%p0 B_2$, and the instruction $\text{branch } \%p2 B_7$?

2. What is the influence region of $\text{branch } \%p0 B_2$ and the influence region of $\text{branch } \%p2 B_7$?

**In search of the Holy Grail** Given the concepts that we have introduced in the last section, we are now ready to determine the set of divergent variables in a Cuda program. With this purpose, we shall rely on Theorem 3.15, which we state without proving:

**Theorem 3.15** **Divergent Variable** A variable $v \in P$ is divergent if, and only if, one of these conditions hold:

1. $v = \text{tid}$ – or any other variable that is particular to a given thread by definition, like $\text{laneId}$.

2. $v$ is defined by an atomic read-modify-write instruction, e.g: $\text{atomic} \{ \ v = *a; \ *a = *a + b \ \}$.

3. $v$ is data dependent on some divergent variable.

4. $v$ is sync dependent on some divergent variable.
Question 3.16 The first item of Theorem 3.15 is the easy part: why \texttt{tid} is always a divergent variable?

Question 3.17 The second item of Theorem 3.15 is not hard to understand either. Explain the semantics of \texttt{atomic \{ v = \*a; \*a = \*a + b \}} and why it leads to divergent \texttt{v}'s. Remember that the instruction will be executed by multiple threads.

Given a program \( P \), a variable \( v \in P \) is data dependent on a variable \( u \in P \) if \( P \) contains some assignment instruction that defines \( v \) and uses \( u \). For instance, in Figure 37 we have that \( \%i \) is data dependent on \( \%i1 \), because of the \( \phi \)-function in block \( B_1 \). We also have that \( \%i1 \) is data dependent on \( \%i \), because of the assignment \( \%i1 = \%i + 1 \) in block \( B_2 \). The problem of determining the transitive closure of the set of divergent variables – given the propagation of data dependences only – is a type of program slicing [40], and it can be solved by a simple graph traversal algorithm. We define the data dependence graph as follows:

- For each variable \( v \in P \), let \( n_v \) be a vertex of \( G \).
- if \( P \) contains an instruction that defines variable \( v \), and uses variable \( u \), then we add an edge from \( n_u \) to \( n_v \).
In order to find the set of divergent variables of $P$, we start from $n_{tid}$, plus the nodes that represent variables defined by atomic instructions, and mark every variable that is reachable from this set of nodes.

It all seems pretty neat, but there is one puzzle piece that we are missing: the so called sync dependencies, which we define below:

**Definition 3.18** Given an instruction **branch** $\%p B$, we say that $\%v$ is sync dependent on $\%p$ if, and only if, the value of $\%v$ at the immediate post-dominator of the branch depends on the outcome of $p$.

**Question 3.19** Just before we move on, take a look into Figure 37 and tell me: which variables are sync dependent on $\%p_0$, $\%p_1$ and $\%p_2$?

If you have pried a good answer out of Question 3.19, then you probably already know how to find the set of variables that are sync dependent on a given branch instruction. Nevertheless, Theorem 3.20, which again we state without proving it, gives a systematic way to determine the set of sync dependent variables in a program.

**Theorem 3.20** Let **branch** $\%p B$ be a branch instruction, and let $l_p$ be its immediate post-dominator. A variable $v$ is sync dependent on $p$ if, and only if, $v$ is defined inside the influence region of the branch and $v$ reaches $l_p$.

Theorem 3.20 gives us a very natural way to handle sync dependences: we can transform them into data dependences using an old intermediate representation called Gated SSA-form [30]. The main point is that only variables produced by $\phi$-functions may diverge due to sync dependent variables, and there is a technique to determine the set of $\phi$-functions affected.

**Question 3.21** Why only variables produced by $\phi$-functions may diverge due to sync dependent variables?

To build the gated SSA-form representation, we augment some $\phi$-functions with predicates, and create some new $\phi$-functions. We say that a $\phi$-function augmented with a predicate is gated. For instance, below we have a $\phi$-function gated by predicate variables $p_1$ and $p_2$:

$$ v = \phi(v_1, \ldots, v_n), p_1, p_2 $$

We give an algorithm to build gaited SSA-form in Figure 38, for a more efficient algorithm, we recommend the work of Tu And Padua [39].

Figure 39 shows the results of running the algorithm from Figure 38 on the program in Figure 37. The $\phi$-functions at $B_4$ and $B_8$ were marked by step (1.a) of our algorithm. Variable $\%j_3$ is sync dependent on variable $\%p_1$, and Variable $\%x$ is sync dependent on variable $\%p_2$. The arity one $\phi$-function at $B_5$ was created by step (1.b.i) of Algorithm 38.

Once we have gated the $\phi$-functions the program’s data dependence graph will subsume the effects of sync dependences. That is, by creating an instruction $v = \phi(v_1, \ldots, v_n), p_1, \ldots, p_k$, ...
Algorithm 38: build gated SSA-form – For each instruction branch $pB$ with an immediate post-dominator $l_p$, we do:

1. For each variable $v$, defined in $IR(p)$, reaching $l_p$ we do:
   (a) if $v$ is used in $l_p$ as a parameter of a $\phi$-function $v' = \phi(\ldots, v, \ldots)$, gaited or not, then we replace this instruction with a new $\phi$-function gaited by $p$.
   (b) if $v$ is used by an assignment instruction $x = f(\ldots, v, \ldots)$, at $l_p$ or at some label $l_x$ which $l_p$ dominates, then we perform the following actions:
      i. we split the live range of $v$, inserting an instruction $v' = \phi(v, \ldots, v), p$ at $l_p$, with a parameter for each predecessor of $l_p$;
      ii. we rename every use of $v$ to $v'$ at $l_p$, or at any block that is dominated by $l_p$;
      iii. we reconvert the source program into SSA form, a necessary action due to the renaming performed in the previous step.

Figura 38: The algorithm that gates $\phi$-functions.

we effectively say that $v$ is data dependent not only on $v_i, 1 \leq i \leq n$, but also on the predicates $p_j, 1 \leq j \leq k$. Moving on with our example, Figure 40 shows the graph created for the program in Figure 39.

Surprisingly, we notice that the instruction branch %p1 $B_4$ cannot cause a divergence, even though the predicate %p1 is data dependent on variable %j1, which is created inside a divergent loop. Indeed, variable %j1 is not divergent, although the variable %p0 that controls the loop is.

**Question 3.22** We can prove the non-divergence of %j1 by induction on the number of iterations of the loop. Can you do it?

Nevertheless, variable %j may cause a divergence in the conditional jump branch, %p2, $B_7$, because it is defined by a $\phi$-function which uses %j1. That is, once the threads synchronize at $B_5$, they might have re-defined %j1 a different number of times. Although this fact cannot cause a divergence inside the influence region of %p0, as Algorithm 38 shows, divergences might happen outside the loop controlled by %p0. Thus, we split the live range of %j outside the loop, with the $\phi$-function in $B_5$, as Figure 39 shows.

**Vivaldi’s Four Seasons Symphony** Once we have been able to distinguish divergent variables from the non-divergent ones, the next step we want to take is to use this information to optimize programs. There exists some compiler optimizations that take benefit from divergence analysis, namely:

**thread reallocation** consists in re-grouping divergent threads in different warps, so that more threads will take similar branching decisions. For instance, each GPU warp
contains 32 SIMD threads. In face of divergences, one can re-group these threads, so that only one warp contains divergent threads. Zhang et al. [43] have implemented this
reallocation by manually inserting shuffling code before divergent branches, obtaining performance gains of up to 37% in some CUDA applications. Similarly, Fung et al. [17] have proposed a way to perform this reallocation at the hardware level, inferring, via simulation, speed-ups of up to 20%.

**Variable sharing** consists in storing non-divergent variables into the shared memory. That is, once we know that a variable \( v \) is non-divergent, we can remove it from one register in each thread, and store it into the shared memory, which has very fast access. This is advantageous because CUDA divides a fixed number of registers among the co-existing threads; thus, the demand for local storage limits the number of active processing elements. This optimization has the effect of reducing the register pressure at each thread; hence, allowing the coexistence of more of them. Collange et al. [11] have designed a hardware mechanism that dynamically detects non-divergent variables and promotes them to shared memory. This technique identifies about 19% of the values read into local registers as non-divergent.

**Branch fusion** is an optimization that attempts to merge two divergent paths as much as possible, in order to maximize the amount of work that threads can do together. There are many ways to implement this optimization. We will describe a technique that is based on the Smith-Waterman gene sequencing algorithm that is very used in Bioinformatics.

**Barrier elimination** consists in removing barriers from branches that the divergence analysis has proved to be non-divergent. This optimization is not likely to produce gains in CUDA programs, for the Nvidia’s implementation is very good at synchronizing threads at the hardware level. Nevertheless, barrier elimination is very effective in hardware where the synchronization of divergence threads is expensive. Additionally, this is the simplest optimization in the lot.

**Trip to the Smurf Village**  Smurfs are tiny blue goblins, about the size of three apples piled together. They wear white trousers and white beanies, except Papa Smurf, who shows off a scarlet red garment. The Smurfs live in a Smurf village, where they share everything, but the houses. Each Smurf has a house, which is a small mushroom, with a door, a chimney, and tiny windows. One day the Smurfs decided to install a TV system in the Smurf Village. Each Smurf got his own antenna, and a small, highly customized TV set. The Smurfs became truly fond of TV, but they started having problems with power: the 99 TV sets were sucking up too much electricity. This nuisance was solved by the Brainy Smurf, who realized something quite interesting: they Smurfs would always watch the same TV channel, never switching to anything else. So, Brainy proposed them to buy a bigger TV set, which would be placed on a communal area in the village. Once all the Smurfs started using only the shared TV, they never again had problems with electricity in the Smurf Village, which was again, a very happy place.

Key to Brainy’s solution was the understanding that all the Smurfs always watched the same programs. That is, they behaved as SIMD threads. Similarly, when the divergence
analysis proves that a variable is non-divergent, then this variable can be shared among every thread. Thus, instead of keeping this variable in the local register bank of every thread, we can move it to the shared memory. This optimization is advantageous when the number of registers is limiting the number of threads simultaneously alive. If you recall from Section 2 (Register to remember), the GPU has a fixed amount of registers, which will be divided among the threads. For instance, the GPU can run up to 768 threads, which will have access to 8,192 registers. This gives to each thread at most 10 registers. An application that requires more than 10 registers per threads will not be able to use the hardware in its full capacity.

As an example, consider Figure 41, which shows a Cuda kernel that computes one of the roots of a second-degree equation, e.g., $ax^2 + bx + c = 0$. Our kernel reads three input arrays, each one with a sequence of coefficients, plus an argument posRoot, which determines which root is the valid solution. This kernel is register intensive, for there are many variables that we would like to keep in registers. Yet, argument posRoot is a non-divergent variable, and we could keep it in the shared memory. If we do it, then we reduce the register pressure in the kernel by 1. Figure 42 shows the kernel that results from this sharing.

```
__global__
void slvEq2(int posRoot, float* aa, float* ba, float* ca, float* root) {
    float a = aa[threadIdx.x];
    float b = ba[threadIdx.x];
    float c = ca[threadIdx.x];
    float delta = b * b - 4 * a * c;
    float sqrtDelta = sqrt(delta);
    if (delta >= 0) {
        if (posRoot) {
            sqrtDelta = - sqrtDelta;
        }
        root[threadIdx.x] = (-b + delta) / 2*a;
    } else {
        root[threadIdx.x] = NAN;
    }
}
```

Figure 41: A Cuda kernel that computes one of the roots of a second-degree equation $ax^2 + bx + c = 0$.

**Question 3.23** Take a look into the kernel in Figure 4. Which variables could you move to the shared memory?
void slvEq2(int posRoot, float* aa, float* ba, float* ca, float* root) {
    __shared__ int sharedPosRoot = posRoot;
    float a = aa[threadIdx.x];
    float b = ba[threadIdx.x];
    float c = ca[threadIdx.x];
    float delta = b * b - 4 * a * c;
    float sqrtDelta = sqrt(delta);
    if (delta >= 0) {
        if (sharedPosRoot) {
            sqrtDelta = -sqrtDelta;
        }
        root[threadIdx.x] = (-b + delta) / 2*a;
    } else {
        root[threadIdx.x] = NAN;
    }
}

Figura 42: The kernel that we obtain from Figure 41 by sharing the non-divergent variable posRoot.

I dream about a world without barriers! The implementation of SIMD conditional branches is substantially complicated by the necessity to handle divergences. However, this complication should not be imposed on non-divergent branches. Thus, many SIMD architectures provide two conditional jumps: one that makes provisions for divergences; and another – simpler – conditional branch that neglects divergences, being only correct in face of non-divergent behavior. As an example, PTX provides bra.uni, a conditional jump that can be used in the absence of divergences. The PTX programmer’s guide says the following about this instruction ⁶:

“All control constructs are assumed to be divergent points unless the control-flow instruction is marked as uniform, using the uni suffix. For divergent control flow, the optimizing code generator automatically determines points of re-convergence. Therefore, a compiler or code author targeting PTX can ignore the issue of divergent threads, but has the opportunity to improve performance by marking branch points as uniform when the compiler or author can guarantee that the branch point is non-divergent.”

Given the results of divergence analysis, implementing the elimination of implicit barriers is only a matter of adding the uni suffix at the end of every branch that uses a non-divergent predicate. Going back to Figure 37, we could apply this optimization on the instruction

⁶PTX programmer’s manual, 2008-10-17, SP-03483-001_v1.3, ISA 1.3
branch \%p1 \texttt{B}_4, at the end of basic block \texttt{B}_2. We cannot change the other branches, for they use the divergent predicates \%p0 and \%p2.

One question that you may be asking yourself is how often we can do branch fusion in the context of actual Cuda programs. The answer, of course, depends on the program, but we have performed this experiment on the Nvidia SDK benchmark collection, and found that 25\% of the branches – one fourth of them – can be safely augmented with the \texttt{uni} suffix.

\textbf{Fusion!} If-then-else branches may have a lot of instructions in common. It would be very nice if we could move these common sequences of code into a common program path, so that threads would spend a minimum of time in divergent parts of the program. Branch fusion is a very extensive kind of \textit{redundancy elimination} \cite{26}, whose purpose is to extract common code from divergent program paths. We will use the example in Figure 43 to explain this optimization. We can see that the sequences of instructions in the paths starting at labels \texttt{l}_4 and \texttt{l}_{13} have many operations – and a few operands – in common. Let’s represent a store instruction by \texttt{↑}, and a load instruction by \texttt{↓}. Thus, we can represent the two sequences of instructions in the branch of Figure 43 (a) by $T = \{\bot, \downarrow, \ast, \ast, /, /, \ast, +, \uparrow\}$ and $F = \{\bot, \downarrow, \ast, \ast, /, /, \ast, +, \uparrow\}$. This example raises three questions, which we can state as follows:

1. does it make sense to merge these two branches in order to share redundant work among divergent threads?

2. What is the most profitable way of merging this redundant work? Notice that, depending on the cost model, this question may be too hard to answer. However, if we are only looking for the longest chains of common instructions, then Figure 43 (b) shows a possible answer.

3. Is there a systematic way to merge these two paths, thus producing a program that is semantically equivalent to the original code? In this case, a possible solution is given in Figure 43 (c). Notice that we have used C-like ternary selectors (\texttt{sel}) to choose the operands of merged instructions. An alternative to architectures that do not offer selectors is the classic \textit{if-conversion} transformation \cite{21, 34}.

In the rest of this material we will try to answer each of these questions in turn; however, before we start, perhaps you would welcome the quick warm-up below:

\textbf{Question 3.24} \textit{Take a look into the program in Figure 43 (a). Is the branch instruction really divergent? Can you justify your answer?}

\textbf{Question 3.25} \textit{Comparing the programs in Figure 43 (a) and (c), which implementation would run faster on a single-core machine?}

\textbf{Question 3.26} \textit{Can you correlate the code sequences in Figure 43 (b) with the program in Figure 43 (c)?}
Question 3.27 There are other alignments, different from the one in Figure 43 (b) that have as many matches. Can you find a different alignment, and see if you can come up with the corresponding program?

Question 3.28 Finding the longest chains of common instructions may be a too simplistic approach, for size is not exactly equivalent to work. What would be a better, more realistic cost model to decrease the amount of redundant work among the divergent warp threads?

Not everything that shines is gold It is meaningful to merge only branches that cause divergent threads to be active at separate times. We call these branches “if-then-else’s”, to distinguish them from “if-then” and “while” branches. For instance, the instruction branch %p1 B4 from Figure 37 is a “if-then” branch: had this branch been divergent, then some threads would have to execute label B3; however, the other threads would not have an exclusive code path to process. In this case, no thread performs redundant work. Still in Figure 37, the possibly divergent instruction branch %p0 B2 is a “while” branch. Threads that escape the loop will have to wait for the threads that have iterations to perform. Finally,
branch \%p2 \ B7 is an “if-then-else” branch. In face of a divergence - which may indeed happen according to our divergence analysis - some threads will execute basic block \ B6, while others will execute basic block \ B7. In this case, each group of divergent threads goes over an exclusive code path. We call the “if-then-else” branches unifiable, and we recognize them in the following way:

**Definition 3.29** Unifiable branch  
Let branch \ p B1 be a conditional jump with two successor blocks \ B1 and \ B2. This branch is unifiable if, and only if, there exists neither a path \ B1 \rightarrow B2, nor a path \ B2 \rightarrow B1 in the branch’s influence region.

**Question 3.30** What is the complexity of proving that a branch is in the “if-then-else” category?

**Playing the biologist**  
As you have probably guessed by now, the main complication of performing branch fusion is to decide which instructions we should merge. In general there are many similar programs that perform the same computation, and trying to find the best one may be quite hard. For instance, if we could re-order the instructions inside a program path perhaps it would be easier to obtain a better code sequence to merge. On the other hand, this would make our problem much more difficult.

**Question 3.31** Imagine that you could re-order the instructions inside a divergent program path. What would be the complexity of finding the most profitable instruction sequence to merge, in this case?

To avoid having to deal with the subtleties of Question 3.31, we will not allow ourselves to re-order instructions. Instead, we will work with a simpler problem model, which we call the Bi-Dimensional Instruction Alignment Problem, and that we define below:

**Definition 3.32** Bi-Dimensional Instruction Alignment  
**Instance:** given the following input parameters:

- two arrays of instructions, \ T = \{i_1, \ldots, i_n\} and \ F = \{j_1, \ldots, j_m\};
- a 2-ary scoring function \ s, such that \ s(i, j) is the profitability of merging instructions \ i and \ j;
- a 2-ary scoring function \ c, such that \ c(i, j) is the cost of the selectors necessary to merge \ i and \ j;
- \ b, the cost of an alignment gap, which, in our case, is the cost of inserting a branch into the code.

**Problem:** find an ordered sequence of pairs \ A = \{(x_1, y_1), \ldots, (x_k, y_k)\}, such that:

- if \ (x, y) \in A, then \ 1 \leq x \leq n, 1 \leq y \leq m
- if \ r > s then \ x_r \geq x_s
• if \( r > s \) then \( y_r \geq y_s \)

• \( \sum (s(x,y) - c(x,y)) - b \times G \) is maximum, where \( (x,y) \in A \), and \( G \) is the number of gaps in the alignment.

In essence, the instruction alignment sequence is interested in computing the most profitable chains of instructions in a matrix. Thus, there is a cost of merging each pair of instructions. For instance, if a load instruction takes, say, 100 cycles, then if we merge two of them together, we are saving 100 cycles from one divergent path. However, in order to perform this merging we may have to insert some selectors in our program. The selectors are not free, but, if they are cheap enough, then the bargain is profitable. We can also merge instructions that are not necessarily the same. For instance, we could merge a comparison for “less-than” with a comparison for “greater-than”, as we can easily convert one instruction in the other. Similarly to traditional redundancy elimination algorithms, we can abuse the power of identities between instructions to maximize the amount of code that we can extract from branches. Actually, it is possible to take this approach to the extreme, employing equality saturation [38] to find the most similar code sequences for each branching path. On the other hand, we cannot merge in any meaningful way an exclusive-or with a division operation. Thus, the cost of pairing these two instructions is very high.

Question 3.33 Can you think about other instructions that, although different, could be merged? You can use more than one instruction to emulate another one.

Question 3.34 There is a small chance that you have never heard about equality saturation. Can you do some research to find out what is this optimization about?

This problem is very similar to gene-sequencing: given two sequences of genes what are the longest chains of common sequences among them? This problem has been very studied before, and it has a very elegant solution: the classic Smith-Waterman’s [35] algorithm for sequence alignment. We will apply the same algorithm to determine the most profitable sequences of instructions to merge. The Smith-Waterman’s algorithm has two steps: first, we build a \textit{profitability matrix} that assigns gains to each possible pairing of instructions. Next, we traverse this matrix, backwardly, in order to discover the best overall instruction alignment.

\textbf{Ebenezer computes his profit} There is a cost, measured in terms of processor cycles, involved in the extraction of a pair of instructions from a divergent path. Figure 44 illustrates the operations necessary to merge two division operations located in different paths of Figure 43 (a). On the left we have the original branch, and on the right the program after branch fusion. The cost that we just mentioned includes selecting the parameters of the new instruction \((2c_s)\), and the execution of the division itself \((c_f)\). We say that the fusion is \textit{profitable} if the cost of executing the merged instruction is less than the cost of executing
the divergent code. We denote this profitability by a scoring function $s$. In the example of Figure 44, we have $s = 2c_j - 2c_s - c_j = c_j - 2c_s$. Notice that if the two variables in the selector are the same, then the selector is not necessary, and its cost is not taken into consideration.

In order to fuse divergent instructions, we must find the most profitable sequence of instructions that can be merged. Our guide in this search is the profitability matrix. To produce the profitability matrix, we write one instruction sequence along the top row, and the other sequence along the leftmost column of a bi-dimensional matrix. Each cell of the matrix is associated to a value $g$, e.g: $H[i, j] = g$, where $g$ is the maximum profit of any possible way to merge instructions up to the indices $i$ and $j$. We compute $H[i, j]$ via the following recurrence relation, where the meaning of the parameters $s, c$ and $b$ is explained in Definition 3.32:

$$H[i, j] = s(i, j) + \text{MAX}\{H[i - 1, j - 1] - c(i, j) - b, H[i, j - 1], H[i - 1, j], 0\}$$

Continuing with our example, Figure 45 shows the profitability matrix that we build to the program in Figure 43 (a). We are using the following scoring function: $s(\downarrow, \downarrow) = s(\uparrow, \uparrow) = 100, s(\ast, \ast) = 2, s(/, /) = 8, s(+, +) = 2$. We assume that the cost of inserting a branch is $b = 2$. Notice that this cost is only payed once, when we leave a sequence of diagonal cells and go to either a vertical or horizontal cell. For the sake of simplicity, we consider $c = 0$. Normally we derive these numbers from the programmers manual. That is, $s(\iota, \iota)$ is the number of cycles that the hardware takes to process instruction $\iota$. The fixed cost $b$ is the number of cycles taken by a selector. The variable cost $c(i, j)$ is the number of cycles taken by a selector, times the number of selectors necessary to merge instructions $i$ and $j$.

Once we have computed the profitability matrix we find a solution to the instruction alignment problem in two steps. First, we scan the matrix to find the cell $H[x, y]$ with the highest profit. Second, we traverse the matrix, starting from $H[x, y]$ and going backwards towards $H[0, 0]$, following the updating direction. That is, if $H[i, j]$ has been updated from $H[i - 1, j - 1]$, then we continue our traversal from the latter cell. In our example, the most profitable cell is $H[9, 8]$, and the most profitable sequence is $A = \langle(1, 1), (2, 2), (3, 3)\rangle$. 

Figura 44: Comparing the costs of divergence and merging.
We have augmented each cell of Figure 45 with the direction of its update. The gray boxes mark the most profitable path in the matrix, which denotes exactly the alignment seen in Figure 43 (c). We give horizontal and vertical updates preference over diagonal updates. That is, when the gain of merging two instructions is the same as the gain of keeping them divergent, we choose the latter. In this way we avoid inserting a branch in the code, if later on we find two instructions that cannot be aligned. For instance, in Figure 45, we could have used $H[1, 3]$ to update $H[2, 4]$ – a diagonal update that effectively merges two multiplications. However, the next two instructions – multiplication and division – have a very low merging score. Thus, we would be forced to insert a branch in the code. On the other hand, given that we have updated $H[2, 4]$ vertically, the cost of the branch had already been payed; hence, we do not pay anything to update $H[2, 5]$ vertically again.

**Question 3.35** Our scoring function has assigned non-zero costs to a few pairs of instructions that are not exactly the same. By looking at Figure 45 can you guess which pairs?

**Question 3.36** What is the complexity of the Smith-Waterman algorithm just described?

**Question 3.37** We must now generate code from the profitability matrix. The process is cumbersome, yet very mechanical. Can you make a correlation between the matrix in Figure 45, and the program in Figure 43? Try thinking about how the gray boxes translate into
code. Do you see a pattern emerging from the diagonal, vertical and horizontal segments in the most profitable path?
The Huckleberry friend at the rainbow’s end

So that is that! I hope you have enjoyed this material, and have learned something about code optimization techniques for GPUs. Code optimization is not exactly a new field of research. On the contrary, since great arch-hacker John Backus and his famous IBM crew [6], a lot of research has been done on the subject. Nevertheless, GPUs, with their unusual execution model, bring many more challenges to compiler writers. If you want to keep up with the novelties, there are a lot of places to go. First of all, I highly recommend the conferences and symposiums. Nowadays it is not difficult to find on-line versions of the most recent papers. There are many computer science conferences that talk about GPUs, both in terms of architecture as well as in terms of programming languages and compiler theory. Here is the list I keep under my pillow:

**PLDI** this is the big place to go if you want to find out about the latest and hottest research in programming languages. I should say that Cuda related stuff has not made much fuzz in PLDI yet, but, little by little, we start seeing works there, like the GPGPU compiler of Yang *et al.* [42].

**POPL** another first tier conference. Works published in POPL tend to be more theoretical than PLDI’s. You will find some very deep papers there, like the Cuda implementation of constraint flow analysis [32].

**PPoPP** this conference is more specialized in parallel programming languages and compilers. There have been many GPU related papers published in this venue, including Ryoo *et al.*’s paper [33] from where I took the examples of matrix multiplication.

**PACT** another well-known conference about tools and compilers for parallel programming. We did a lot of research using Ocelot, an open-source Cuda compiler, which was described in a PACT paper [14].

**MICRO** this is the leading architecture conference. GPUs have been the main subject in a number of papers into this conference. Fung *et al.*’s work, for instance, is a personal favorite [17].

In addition to the conferences, there is a lot of buzz going on around GPUs in the Internet. The Nvidia’s forums are a wonderful place to get answers for everyday programming questions. I also recommend Ocelot’s mailing list, which is growing day by day, and is populated by a very supportive team of programmers and academics.

It will be certainly very interesting to find out where all of this will take us. As almost everything in Computer Science, the limit is people’s creativity, and I could bet big money that there are still many very beautiful optimization algorithms waiting to be found!
Referências


