Master Électronique, Énergie Électrique, Automatique
Parcours Électronique des Système Embarqués

Projet d'Initiation à la Recherche

Mixed-Precision in Graphics Processing Units

Author :
M. Quentin GALLOUEDEC

Teachers :
M. Ian O'CONNOR
M. Laurent QUIQUEREZ

March 5, 2020

Abstract : Modern graphics computing units (GPUs) are designed and optimized to perform highly parallel numerical calculations. This parallelism has enabled (and promises) significant advantages, both in terms of energy performance and calculation.

In this document, we take stock of the different applications of mixed precision. We recall the standards currently used in the overwhelming majority of systems in terms of numerical computation. We show that the mixed precision which decreases the precision at the input of an operation does not necessarily decrease the precision of its output. We show that this previous principle allows its transposition into one of the branches that most needs computing power: machine learning. The use of fixed point numbers and half-precision are two very effective ways to increase the learning ability of complex neural networks. Mixed precision still requires the use of suitable hardware, failing which the calculation time could on the contrary be lengthened. The NVIDIA Tensor Core that is found among others in their Tesla V100 range, is an example of implementation at the hardware level of mixed precision. On the other hand, by abandoning the traditional von Neumann model, mixed precision can also be transposed to a lower level of abstraction, using phase change memories.

Résumé : Les GPUs modernes sont conçus et optimisés pour réaliser des calculs numériques hautement parallèles. Ce parallélisme a permis (et promet) des avantages significatifs, autant en matière de performances énergétiques que calculatoire.

Dans ce document, nous faisons le point sur les différentes applications de la précision mixte. Nous rappelons les standards actuellement utilisés dans l'écrasante majorité des systèmes en matière de calcul numériques. Nous montrons que la précision mixte qui diminue la précision en entrée d'une opération ne diminue pas nécessairement la précision de sa sortie. Nous montrons que ce précédent principe permet sa transposition dans l'une des branches qui a le plus besoin de puissance de calcul : la machine learning. L'utilisation de nombres à virgules fixe et de la demi-précision sont deux moyens très efficaces d'augmenter la capacité d'apprentissage de réseaux neuronaux complexes. La précision mixte nécessite tout de même d'utiliser un hardware adapté, à défaut de quoi le temps de calcul pourraient être au contraire allongé. Le tensor Core de NVIDIA que l'on trouve entre autres au sein leur gamme Tesla V100, est un exemple d'implémentation au niveau matériel de la précision mixte. D'autre part, en abandonnant le modèle traditionnel de von Neumann, la précision mixte est également transposable à un niveau d'abstraction plus bas, en utilisant des mémoires à changement de phase.

1 Introduction

Low-precision floating point numbers use fewer bits than high-precision floating point numbers. As a result, they are subject to larger rounding errors. Therefore, the error caused by rounding can have a large influence on the total error. Some algorithms that use simple precision could not handle the error induced by this decrease in precision. Nevertheless, in many cases, it would be very beneficial to reduce the precision of floating numbers to gain both speed and power. Clustering or graph ranking algorithms or the training of dense neural networks are some examples. The use of mixed precision may well be a solution to reduce size, power consumption, weight and speed in many computer and electronic applications.

In the first part of this paper, we will first study the IEEE 754 standard that defines the format of the binary representation of real numbers. We will then deduce some theoretical principles fundamental to the use of mixed precision. In the second part, we will study one of the applications that can benefit the most from the development of mixed precision: machine learning. We will see how the chosen binary representation is at the heart of the performance of its algorithms. Finally, in a last part, we will see the current research axes to dimension the hardware to take full advantage of the application of mixed precision.

2 IEEE 754 : Standard for Floating-Point Arithmetic

The vast majority of computers use *floating-point arithmetic* to represent real numbers. The technical standard for floating-point arithmetic was established in 1985 by the IEEE Standard for Floating-Point Arithmetic (IEEE 754)[1]. Since memories use the binary form, there is necessarily a loss of precision in the transition from real numbers to floating point numbers.

2.1 Single-precision floating-point format

The single-precision floating-point format uses 32 bits of memory. This is why it is called `float32`. A half-precision floating-point number requires 16 bits of memory, and a double-precision floating-point number uses 64 bits of memory. Throughout this paper, we will use `float16`, `float32` and `float64` to refer to a half-, single- and double-precision floating-point number respectively.

Here is the binary representation of a `float32`. The encoding principle is similar for the encoding of `float16` and `float64`

$$\underbrace{\texttt{X}}_{\text{sign (1 bits)}}\ \underbrace{\texttt{XXXXXXX}}_{\text{exponent (8 bits)}}\ \underbrace{\texttt{XXXXXXX}\cdots\texttt{XXXXXXX}}_{\text{fraction (23 bits)}}$$

Next, let's index the binary values like this.

$$\underset{b_{31}}{\texttt{X}}\ \underset{b_{30}}{\texttt{X}}\ \underset{b_{29}}{\texttt{X}}\ \cdots\ \underset{b_1}{\texttt{X}}\ \underset{b_0}{\texttt{X}}$$

Let us denote by $\mathbb{B}^{32}$ the set of 32 boolean sequences. Thus, $(b_i)_{\mathbb{N}_{<32}} \in \mathbb{B}^{32}$ is the binary representation of the value.
Let us set the following values :

$$s = (-1)^{b_{31}} \tag{1}$$

$$e = \sum_{i=0}^{7} b_{23+i} 2^i \tag{2}$$

$$f = \sum_{i=1}^{23} b_{i-23} 2^{-i} \tag{3}$$

Let us denote by $\bar{\mathbb{R}}$ the set of real numbers to which we add the $\{-\infty, +\infty\}$. Thus, the function $\mathcal{F}^{32} : \mathbb{B}^{32} \longrightarrow \bar{\mathbb{R}}$ that associates the binary representation with its corresponding real number is[1] :

[1] You might notice that the function is not an injection, since 0 and $\pm\infty$ can be coded in several different ways.

$$\mathcal{F}^{32} : (b_i)_{\mathbb{N}_{<32}} \mapsto \begin{cases} s \times 2^{-127} \times f & \text{if} \quad e = 0 \\ s \times 2^{e-127} \times (1+f) & \text{if} \quad e \in [\![1, 254]\!] \\ s \times +\infty & \text{if} \quad e = 255 \end{cases} \tag{4}$$

For $e = 0$ the numbers are called *sub-normal numbers*.

Examples

`0 00000000 00000000000000000000000`
$= (-1)^0 \times 2^{-127} \times 0 = 0$

`1 10000000 00000000000000000000000`
$= -1 \times 2^{128-127} \times (1+0) = -2$

`0 10000011 10000000000000000000000`
$= 1 \times 2^{131-127} \times (1+2^{-1}) = 12$

`0 01111111 00000000000000000000000`
$= 1 \times 2^{127-127} \times (1+0) = 1$

The smallest value bigger than 1 :

`0 01111111 00000000000000000000001`
$= 1 + 2^{-23} \approx 1.00000011920929$

We denote F^{32} as the image of $\mathcal{F}^{32}$ under $\mathbb{B}^{32}$. The density F^{32} is not uniform over $\mathbb{R}$. But, in every interval between two power of 2, there are the exact same amount of numbers.

$$\forall n \in [\![-126, 127]\!] \quad \#F^{32}_{[2^n, 2^{n+1}[} = 2^{23} \tag{5}$$

Where $F^{32}_{[2^n,2^{n+1}[} = F^{32} \cap [2^n, 2^{n+1}[$

The figure 1 shows a naive representation of F^{32} density within $\mathbb{R}$.

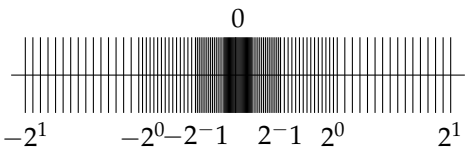

Figure 1: Naive representation of F^{32} density within within $\mathbb{R}$.

2.2 Other formats

Other formats are defined in the standard. The table 1 describes some features of these formats[2].

Size (bits)	Name	e size (bits)	f size (bits)
16	Half	5	10
32	Single	8	23
64	Double	11	52
128	Quadruple	15	112
256	Octuple	19	236

Table 1: Formats of IEEE Standard for Floating-Point Arithmetic

Number encoding for these formats follows the same rules as for `float32`. The only parameter that differs is the size of the bit transposition of the encoded numbers.

Extended precision formats are also defined in the standard. It allows a greater precision than the basic floating point formats. It uses 40 bits or 80 bits. It does not encode numbers exactly the same way : it uses also a bit for the integer part. Since it is rarely used at the code level, we will focus on binary floating-point formats.

2.3 Mixed precision theory

Mixed precision corresponds to a calculation method that uses different levels of precision for the same operation. The aim is to benefit from the shorter calculation time of coarse precision, while maintaining the accuracy of the finer precision. There are several methods of applying the mixed-precision principle.

One of them is to start the calculations start with `float16` values for rapid matrix math. But as the numbers are computed, the machine stores the result at a higher

[2] Actually, there are also decimal formats described in the IEEE Standard for Floating-Point Arithmetic. Since it is less used, we won't describe it.

precision. For instance, if multiplying two 16-bit matrices together, the answer is 32 bits in size. By accumulating the answers, the accuracy becomes finer and finer, until it reaches a level of precision equivalent to those obtained using `float64`. Since calculations are made with `float16`, they are faster, less memory is used and power consumption is also lower. This operation is called Fused Multiply and Add (FMA).

The FMA operation compute operations like $x \times y + z$ (where x, y and z are floating-point numbers) as a single floating point operation. The classical approach would be to first perform $x \times y$, round, then add the result with z, and round again. The FMA calculates $x \times y + z$ at once, then rounds. Thus, the FMA is faster, and more accurate. On the Itanium processor, the FMA operation requires the same number of cycles as multiplication or addition [2].

Since the product of two matrices is equivalent to making sums of products (see equation 6), the FMA operation is particularly suitable. The coefficients of the two matrices $(a_{i,k}, b_{k,j})$ can be in `float16`, while the result $(c_{i,j})$ will be in `float32`.

$$\forall (i,j) \in \mathbb{N}_m \times \mathbb{N}_p \quad c_{i,j} = \sum_{k=0}^{n} a_{i,k} b_{k,j} \qquad (6)$$

Where $m, n, p \in \mathbb{N}$, $(a_{i,j}) \in M(m,n)$, $(b_{i,j}) \in M(n,p)$, et $(c_{i,j}) \in M(m,p)$ the result of $(c_{i,j}) = (a_{i,j}) * (b_{i,j})$.

3 Main application : machine learning

Training a large neural network requires the ability to perform a large number of operations per second. Recent work on the CIFAR-10 database has reduced the misclassification rate to below two percents [3, 4, 5]. The best classifier uses 557 million parameters. The top-3 neural networks in this dataset all use at least 10 million parameters.

In the following sub-sections, we will discuss some proposed methods for using different representations during the training phase. These methods aim to increase the computing speed and decrease the memory used while maintaining the training performance.

3.1 Using fixed-point numbers with stochastic rounding

Deep neural networks can be trained using 16-bit fixed-point number instead of `float32`. Some explaination of fixed-point number represenation is given in the appendix A. By using stochastic rounding, there is hardly any degradation in the classification accuracy [6]. In the following lines we explain the results of some work on this subject.

Using fixed-point numbers requires a conversion stage to go from floating-point represenation to its new representation. Let us denote `FL` as the number of fractional bits in the fixedpoint represenation, and $\epsilon = 2^{-\mathrm{FL}}$ the gap between two numbers. For a real number x, we denote $\lfloor x \rfloor$ as the largest multiple of ϵ less than or equal to x. We consider the two following rounding scheme:

- Round to nearest : the rouding of a given real number x is set to minimize the distance between this number and it's rounding.
- Stochastic rounding : the rounding of a given real number x can be either the nearest higher value or the nearest lower value. The probability that its rounding is the value just below is $1 - \frac{x - \lfloor x \rfloor}{\epsilon}$.

Several trainings were done on the MNIST dataset using fully connected Deep Neural Networks (DNNs). These trainings compared the two rounding methods by varying the length for fixed-point numbers. The control training is the one using `float32`. By using stochastic rounding, the loss decreases in the same way for `float32`, 14-bit, 10-bit and 8-bit fixed-point numbers. It means that the

neural network trains just as good, whether it uses float32 or 8-bit fixed-point numbers computations. Thus, it allows to use less memory, less power, and compute faster.

3.2 Using float16 instead of float32

Another way for training deep neural networks is using float16 instead of float32. Since float16 use half as much memory, the memory requirement can be halved. However, the use of float16 can result in the loss of essential information. To avoid this, and to obtain results as good as those obtained with float32, some methods have been proposed [7].

3.2.1 Store float32 model to preserve small weights

While training and updating float16 weights during the backward propagation, many weight are bound to become very small. The shortest float16 possible positive number is $2^{-14} \times 2^{-10} = 2^{-24} \approx 5.96 \times 10^{-8}$ (see section 2.2). If the weight is below, is will be set to 0. Small weights have been shown to make a significant contribution to the learning capacity of networks ([7] figure 2a). One way to prevent the disappearance of the small weights is to store a float32 copy of the model. This copy would accumulate the gradient after each optimizer step. The results obtained with storage of a copy of the weights prove to be much better than those obtained without storage. Actually, these results are as good as those obtained with float32 only.

3.2.2 Scalling the loss

The loss is calculated for each batch to prepare the backpropagation. Since the main purpose of a classification problem is to minimize the loss, the loss will decrease and reach very low values. During a float32 training of Multibox SSD detector, it was observed that the network activation gradient values are mostly below 2^{-24}[8], which is the smallest value that float16 can encode. Since almost no value exceeds 2^{-8}, scaling up the gradients allows them to occupy more of the float16 representable space[3] Scalling the gradient by a factor a 2^4 is enough to be obtain the same accuracy as float32 training.

3.2.3 Accumulating partial products into an float32 value

The training of neural networks requires a limited number of different operations. For each operation, it is possible that the result may not be completely accurate. The result will often be the closest number to the result that can be encoded in the chosen standard. A computational inaccuracy is shown in the python command lines below[4].

```
>>> "%.20f" % (1.0/10)
'0.10000000000000000555'
```

Code Listing 1: Computationnal innacurracy example in Python code

The smaller the encoding size, the greater the innacuracy. To maintain a proper accuracy, some networks need that float16 vector dot-product accumulates their partial products into an float32 value before storing it into the memory [7]. In doing so, the advantage of using reduced precision is partially lost. However, the latest Graphic Processing Units (GPUs) such as Nvidia's Tesla natively perform these operations. This allows a considerable increase in speed while maintaining the advantages of working with float16 [9].

[3] When training neural network, you can have another gradient related issue : gradients can explode when bakpropopagating. This is when they get exponentially large from being multiplied by numbers larger than 1. One method of preventing this problem is the *gradient clipping*. It will clip the gradients between two numbers to prevent them from getting too large.

[4] Python's standard float type is a float64.

4 Hardware

Since the calculation units are generally designed to operate on double precision floating point numbers and on integers, the use of mixed precision may not be optimal because of the conversion stage required to go from one precision to another. In the following sections we will detail recent efforts in terms of hardware to adapt the computing units to mixed precision.

4.1 Matrix computing on GPU

In its original design, the GPU was sized to quickly manipulate memory to speed up image creation and processing. The resulting buffer memory was intended to be sent to the display device.

The main difference between them and a Central Process Unit is their highly parallel structure. This makes GPUs much more efficient in terms of operating on data that can be processed in parallel. This is especially the case for matrix calculations. Each component of the matrix resulting from an operation depends only on the coefficients of the matrix argument. They can thus all be calculated in parallel.

4.2 Half-precision without custom hardware

We have seen that, with the right architecture, mixed precision can be a very efficient way to increase the computing speed while maintaining the accuracy of `float32`. However, most of the processing units were not designed for `float16` computation. In C language, `float16` does not exist natively. However, a library can be used to process `float16` calculations[5]. Let us denote $n \in \mathbb{N}$ a natural integer, $A, B \in M_n(\mathbb{R})$. We want to estimate the time needed to complete the product $A \times B$ for `float16`, `float32` and `float32` format. The C++ code used, and the main characteristics of the computer used are available in appendix B. By varying the size of the matrices, the obtained results are presented in the figure B.

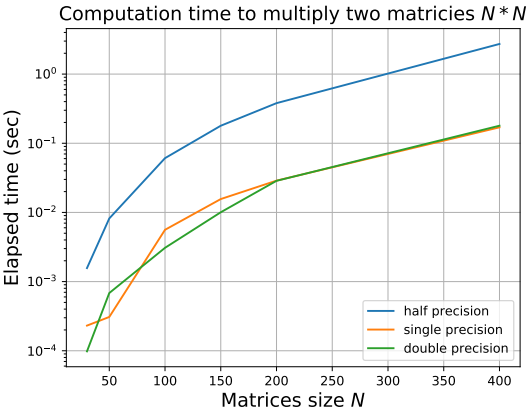

Figure 2: Computation time for a multiplication of two matrices of size $N \times N$ using `float16`, `float32` and `float64`

The figure shows a counterintuitive result. `float32` and `float64` need about the same time to compute a product of matrices, irrespective of their size. What is more, the time needed when using `float16` is ten times greater than the latter. The reason for the previous two results is that the processor I used to do these calculations does not natively support `float16` operations. The processor converts each input to `float64`, performs the calculation, and then converts the output back to `float16` or `float32` (depending of the entry format) [10, 11]. In some cases, the use of `float16` can be an advantage. Since the conversion is done directly at the core level, the smaller the size of the data, the greater the capacity to store in cache memory.

4.3 Tensor core

The first specialized units using FMA operations to make a product of 4x4 arrays per clock cycle were introduced by the Volta version of NVIDIA GPUs. Using mixed precision, the NVIDIA Tesla V100 accelerator (featuring the Tesla V100 microarchitecture)

[5] Available at : `http://half.sourceforge.net/half_8hpp.html`

reaches 125 Tflops/sec. In the following lines, we study how this performance is achieved, and quantify the loss of precsion induced by the use of mixed precision.

Figure 3 shows a simplified schematic of the Volta SM architecture. It shows the tensorcore, which comes in addition to the cores dedicated to float and integer operations.

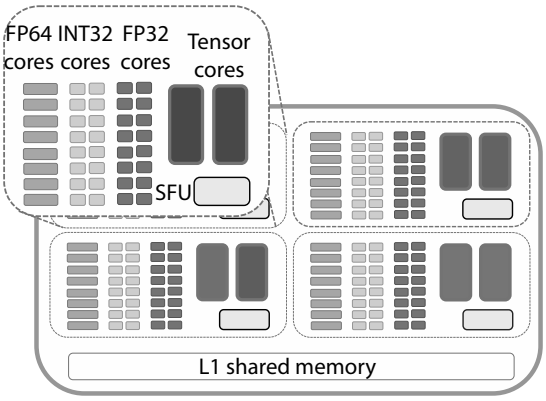

Figure 3: Simplified diagram of the Volta SM architecture. The NVIDIA Tesla V100 uses 80 SMs. [12]

In practice, the tensor cores have been able to deliver up to 83 Tflops/s in mixed precision[6].

4.4 In-memory mixed precision

Traditional von Neumann architecture reaches its limits. Solutions such as in-memory computing, first introduced in 2012 [13], can be used to improve performance in terms of computing power. The spatial separation between the storage unit and the computing unit is one of the main contributions of computing time. The cache memory of the processors aims at reducing this distance, by selecting the data that will certainly be needed for a next calculation, and keeping them close to the Arithmetic Logic Unit (ALU). In-memory goes even further, by processing and storing computational data on the same physical devices organized in a computational memory unit. It uses nanoscale resistive memory devices within a computational memory unit. These units are used for both processing and memory. The figure 4 shows a view of an example of resistive memory.

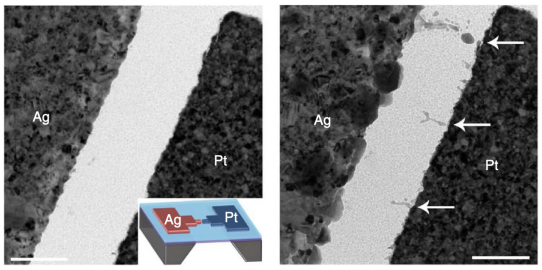

Figure 4: TEm image of an as-fabricated SiO2-based resistive memories. scale bar: 200 nm [14]

A hybrid system has been studied, in which a von Neumann machine and a computational memory unit coexist [15]. The calculation memory unit performs the coarse part of a calculation, and the von Neumann machine implements a backtracking method to iteratively improve the accuracy of the result. It can be defined as a mixed-precision in-memory computing. The goal of this hybridization is to combine the high precision of digital computing with the energy efficiency of in-memory computing.

Phase-Change Memory (PCM) are resistive memory devices that can be programmed to get a specific conductance value. This value is reached by changing the configuration of the amorphous and crystalline phase within the device. It exploit the behaviour of chalcogenide glass. One way to make the glass amorphous is to change the coordination state of the Germanium atoms with a laser pulse [16]. One million of these devices have been implemented in a prototype chip.

To test the performance of the latter, the chosen case study is the multiplication of a matrix by a vector. First, let β_n , γ_n be numbers generated uniformly in $[0, 1]$, And $\theta_n = \beta_n \gamma_n$. Let $\hat{\theta}$ be the averaged result on K PCM devices

[6] Measurement performed on a Tesla V100 GPU.

used. The calculation is performed 1024 times. The figure 5 shows the error $\hat{\theta} - \theta$ distribution with different values of K. The standard deviation cis of the order of $K^{-0.5}$, and the mean is 0.

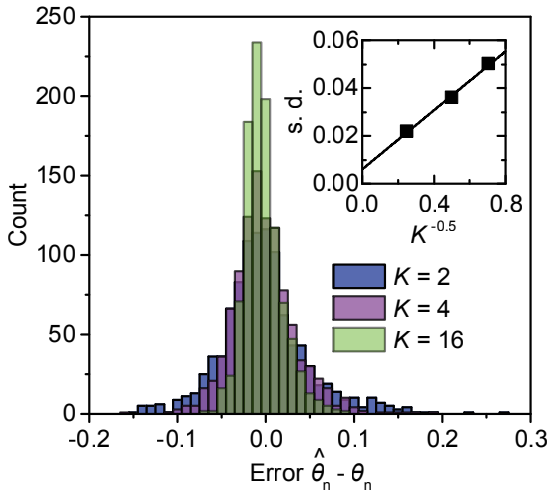

Figure 5: Distribution of scalar multiplication error using $1024 \times K$ PCM devices [16]

By accurately solving a system of 5 000 equations using 998,752 CFM devices, these devices have proven their effectiveness in this case of use.

5 Conclusion and future work

The loss of accuracy due to mixed precision can be an obstacle to the adoption of this calculation method. However, we have shown that with some adaptation, the computing algorithm can benefit greatly from mixed precision. In most cases, it is possible to obtain the same level of precsion as those obtained with the double precsion calculation. NVIDIA is confident that applications that require a large amount of computing capacity are very likely to benefit greatly from using NVIDIA Tensor Cores and mixed precsion. A hybrid system comprising a calculation memory unit was imagined and created. It performs the major part of a given calculation task while a processing unit iteratively improves the result. Today's areas of work for NVIDIA are testing Tensor Cores on applications such as Nek5000 [17] or Fast Multipole Method-accelerated (FFT) [18]. Other works on the hardware aim to transpose this inmemory calculation method for applications other than linear system resolution, e.g. Machine Learning.

Appendices

A Fixed-point numbers

Fixed-point number representation is a data type that represents a finite, fixed number of numbers after the decimal point.

$$\underbrace{\texttt{XXX}\cdots\texttt{XXX}}_{\text{integer part } (m \text{ bits})} \;.\; \underbrace{\texttt{XXX}\cdots\texttt{XXX}}_{\text{fractional part}(n \text{ bits})}$$

Let's index the binary values like this.

$$\underset{b_{n+m-1}}{\texttt{X}} \;\cdots\; \underset{b_n}{\texttt{X}}\texttt{.}\; \underset{b_{n-1}}{\texttt{X}} \;\cdots\; \underset{b_0}{\texttt{X}}$$

Then, the value encoded is :

$$\sum_{i=0}^{n+m-1} b_i 2^{i-n} \tag{7}$$

The density of fixed-point numbers is constant in the real numbers. The gap between two values is always 2^{-n}.

The main advantage of using a fixed-point representation is performance. The value stored in memory is an integer, and the calculation units have very good performance in terms of integer operation.

B Comparison of calculation speeds according to the precision used

The code presented in the listing 2 was used with values of `N` between 30 and 400 and with `TYPE` with values `half`, `float`, `double`. To run the code, you will need download the `half.hpp` header available on `http://half.sourceforge.net/half_8hpp.html`. The computer used to obtain the resuls showed in figure is described in the following lines.

Model : MacBook Pro
Model id : MacBookPro15,4

Processor Name : Quad-Core Intel Core i5
Processor Speed : 1,4 GHz
Number of Processors : 1
Total Number of Cores : 4
L2 Cache (per Core) : 256 Ko
L3 Cache : 6 Mo
Hyper-Threading Technology : Activé
Memory : 8 Go
Compiler : Apple clang version 11.0.0 (clang-1100.0.33.16)
Target : x86_64-apple-darwin19.2.0
Thread model: posix

```cpp
#include "half.hpp"
#include <iostream>
#include <ctime>

using half_float::half;

#define N 150 // Choose matrices size
#define TYPE half // Choose type

int main()
{
  // Define the matrices
  TYPE A[N][N];
  TYPE B[N][N];
  TYPE C[N][N];

  // Define matrices coef bounds
  float LO = -128;
  float HI = 128;

  // Variable used for timing
  clock_t end;
  clock_t begin;
  double elapsed_secs;

  // Set random number in A and B.
  srand((unsigned int)time(NULL));
  for(int i=0; i<N; ++i)
    for(int j=0; j< N; ++j)
    {
      A[i][j]=LO+rand()/(RAND_MAX/(HI-LO));
      B[i][j]=LO+rand()/(RAND_MAX/(HI-LO));
    }

  begin = clock();

  // Multiplying A and B.
  for(int i=0; i<N; ++i)
    for(int j=0; j<N; ++j)
      for(int k=0; k<N; ++k)
      {
        C[i][j] += A[i][k] * B[k][j];
      }

  end = clock();

  // Show the elapsed time
  elapsed_secs = double(end-begin)/
    CLOCKS_PER_SEC;
  std::cout << "elapsed secs : " <<
    elapsed_secs << std::endl;

  return 0;
}
```

Code Listing 2: Code used to etimate to time needed to perform the multiplication of two matrices `A` and `B` of size `N`

Notations

$\mathbb{N}$	Set of natural numbers
$\mathbb{R}$	Set of real numbers
$[\![i, j]\!]$	Set of integers between i and j included
$[\![i, j[\![$	Set of integers between i included and j excluded
E^n	Cartesian power of a set E
$\#E$	The cardinality of a set E
$\mathbb{B}$	Boole set : $\{0, 1\}$
$f(E)$	Image set of a set E through a function f
$\bar{\mathbb{R}}$	$\mathbb{R} \cup \{+\infty, -\infty\}$
$M_{n,p}(\mathbb{R})$	Set of real matrices of size $n \times p$ where $n, p \in \mathbb{N}$
$M_n(\mathbb{R})$	$M_{n,n}(\mathbb{R}) \quad n \in \mathbb{N}$
$\mathcal{F}^{32}$	Function from B^{32} to $\bar{\mathbb{R}}$ which gives the real number associated with its floating point representation.
F^{32}	$\mathcal{F}^{32}(B^{32})$
$F^{32}_{[2^n, 2^{n+1}[}$	$F^{32} \cap [2^n, 2^{n+1}]$: all real numbers that have a `float32` represenation

Table 2: Notations

References

[1] IEEE Standard Association, 3 Park Avenue, New York, NY 10016-5997 USA. *IEEE Std 754^{TM}2019 Standard for Floating-Point Arithmetic*, 2019 edition, 2019. Revision of IEEE SId 754-2008.

[2] Stef Graillat, Philippe Langlois, Nicolas Louvet, G Hanrot, and P Zimmermann. Accurate dot products with fma. In *RNC-7, Real Numbers and Computer Conference, Nancy, France*, pages 141–142, 2006.

[3] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112, 2019.

[4] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. A survey on neural architecture search. *CoRR*, abs/1905.01392, 2019.

[5] Ekin Dogus Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation policies from data. *CoRR*, abs/1805.09501, 2018.

[6] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. *CoRR*, abs/1502.02551, 2015.

[7] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. *CoRR*, abs/1710.03740, 2017.

[8] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[9] Tesla NVIDIA. V100 gpu architecture. the world's most advanced data center gpu. version wp-08608-001_v1. 1. *NVIDIA. Aug*, pages 15–16, 2017.

[10] R. Hyde. *The Art of Assembly Language*. ITPro collection. No Starch Press, 2003.

[11] Patrick Konsor. Performance Benefits of Half Precision Floats. Technical report, 08 2012.

[12] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. NVIDIA tensor core programmability, performance & precision. *CoRR*, abs/1803.04014, 2018.

[13] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX.

[14] Yuchao Yang, Peng Gao, Siddharth Gaba, Ting Chang, Xiaoqing Pan, and Wei Lu. Observation of conducting filament growth in nanoscale resistive memories. *Nature communications*, 3(1):1–8, 2012.

[15] Manuel Le Gallo, Abu Sebastian, Roland Mathis, Matteo Manica, Tomas Tuma, Costas Bekas, Alessandro Curioni, and Evangelos Eleftheriou. Mixed-precision memcomputing. *CoRR*, abs/1701.04279, 2017.

[16] Robert E. Simpson, Paul Fons, Alexander V Kolobov, Toshio Fukaya, Miloš Krbal, Takanori Yagi, and Junji Tominaga. Interfacial phase-change memory. *Nature nanotechnology*, 6 8:501–5, 2011.

[17] Nicolas Offermans, Oana Marin, Michel Schanen, Jing Gong, Paul F. Fischer, Philipp Schlatter, Aleks Obabko, Adam Peplinski, Maxwell Hutchinson, and Elia Merzari. On the strong scaling of the spectral element solver nek5000 on petascale systems. *CoRR*, abs/1706.02970, 2017.

[18] Cris Cecka. Low communication fmm-accelerated fft on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2017.