FIRST LINE OF TITLE
SECOND LINE OF TITLE

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the service academique.

Thèse n. 1234 2011
présenté le 12 Mars 2011
à la Faculté des Sciences de Base
laboratoire SuperScience
programme doctoral en SuperScience
École Polytechnique Fédérale de Lausanne

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

pour l'obtention du grade de Docteur ès Sciences
par

Paolino Paperino

acceptée sur proposition du jury:

Prof Name Surname, président du jury
Prof Name Surname, directeur de thèse
Prof Name Surname, rapporteur
Prof Name Surname, rapporteur
Prof Name Surname, rapporteur

Lausanne, EPFL, 2011

Contents

List of Figures

List of Tables

Torsion Part I

Part II Connection

6 Bench for HPC

6.1 Introduction

In this section aims at providing basic but reliable guidlines to produce fast and mannagable code for our algorithms

Most compilers with which you are probably familiar are standalone programs which take as input some source code text and compile it into machine code (or some other target representation).

cach miss : une donnée n'est pas dans le cache

[Dre07] [Aka12]

L1 = 64kB L2 = 512kB L3 = 4096kB

6.2 Languages

- Csharp - Julia - C++ - Intel MKL - OpenBLAS

Parallelization vs. Vectorization (SIMD)

SIMD :

http://www.drdobbs.com/windows/64-bit-simd-code-from-c/240168851

6.3 From syntax to processor

A short story about how a code is translated to get machin instructions

6.4 Data Structure

Array of Structures (AOS) vs Structure of Arrays (SOA)

The most common and likely well-known data structure is the array, which contains a contiguous collection of data items that can be accessed by an ordinal index. This data can be organized as an Array Of Structures (AOS) or a Structure Of Arrays (SOA). While AOS organization is excellent for encapsulation it can be poor for use of vector processing. Selecting appropriate data structures can also make vectorization of the resulting code more effective. To illustrate this point, compare the traditional array-of- structures (AoS) arrangement for storing the r, g, b components of a set of three- dimensional points with the alternative structure-of-arrays (SoA) arrangement for storing this set.

http://hectorgon.blogspot.fr/2006/08/array-of-structures-vs-structure-of.html

http://arxiv.org/pdf/1402.4986.pdf

sqrt | Float64	CPU (ns/el)	Allocation (Bytes)
Allocation	4	8,112
Julia vectorized	9	8,080
Julia broadcast	13	8,352
Julia broadcast!	7	16
Julia map	100	48,000
Julia map!	92	48,000
VML (allocation)	6	8,080
VML (in-place)	4	0

Table 6.1 – Memory allocations for various methods computing sqrt(a) for $n = 10^4$

6.5 Memory allocation and garbage collection

Toutes les syntaxes ne sont pas égales en terme de gestion de la mémoire. Le problème, c'est le passage de la GC qui est couteux en temps. Donc il faut essayer de minimiser l'utilisation mémoire. Idéalement le problème peut rester dans le cache du processeur (mémoire d'accès bcp plus rapide que la RAM). Donc la meilleur stratégie consiste à pré-allouer les tableaux et à faire des opération "in-place" au maximum, c'est à dire d'écraser les donner au fur et à mesure du calcul.

Par ailleurs les accès mémoires sont lents. Plus la taille du problème reste petite, plus le problème peut être résolu en restant dans le cache. (latency).

http://stackoverflow.com/questions/4087280/approximate-cost-to-access-various-caches-and-main-memory

MOST CPU's today uses the memory on multiple level. Generally the memory at the proximity of CPU is costly and less, whereas the memory at the distance (wire distance) is bigger, slower and cheap [1]. Today getting the computer in market with 8GB DRAM is cheap, but L1/L2 cache of such computer is very small in terms of 10's of KB's and few MB's respectively. The access time of L1 (that is generally SRAM) is few cycles whereas L2 is few 10's cycles and accessing main memory is considered a bad programming if accessed too frequently. The access time is huge and in terms of 100's of cycles. So optimizing the code to run and access L1 Instruction and Data cache is the simplest way to optimizing the code.

On remarque que l'allocation mémoire est très différente d'une fonction à l'autre. Il est important de privilégier des opération "in-place" pour contenir l'allocation mémoire, sinon on risque de déclencher la GC qui est couteuse.

Les temps CPU sont indicatifs car le bench est fait sur une durée caractéristique trop courte

Ici on met en évidence la non linéarité du coût d'allocation par élément d'un tableau de taille n. On remarque que la différence entre le coût de sqrt et le coût de l'allocation est constante : c'est le coût de sqrt hors allocation. Attention, cette notion est "language dependent" car les allocations sont gérées par la GC.

Remarque, on trouverait sans doute la même chose pour MKL, à cause du marshalling : le coût d'appel à une fonction C est supérieur à celui d'une fonction managée (cf HPC .Net)

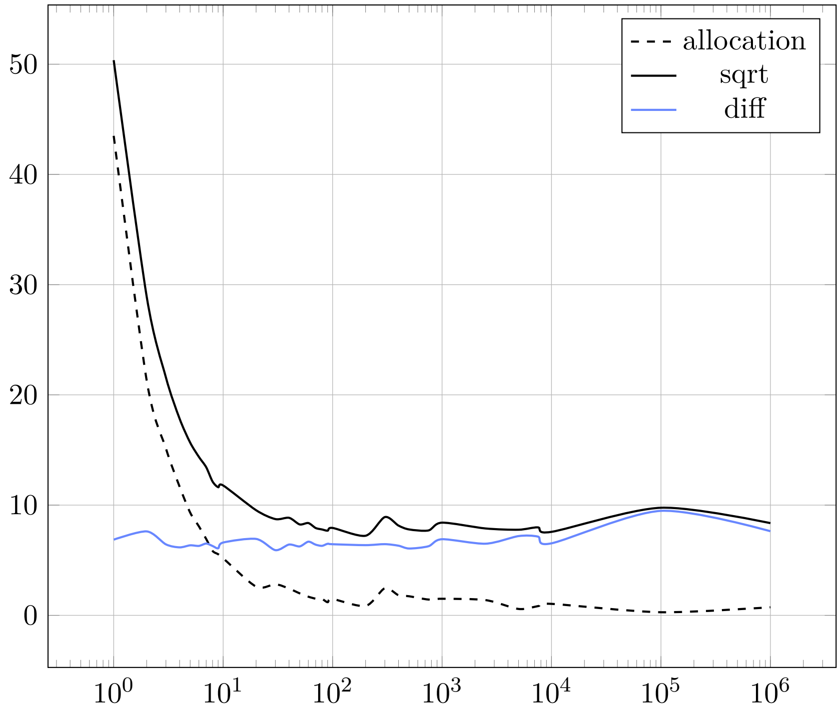
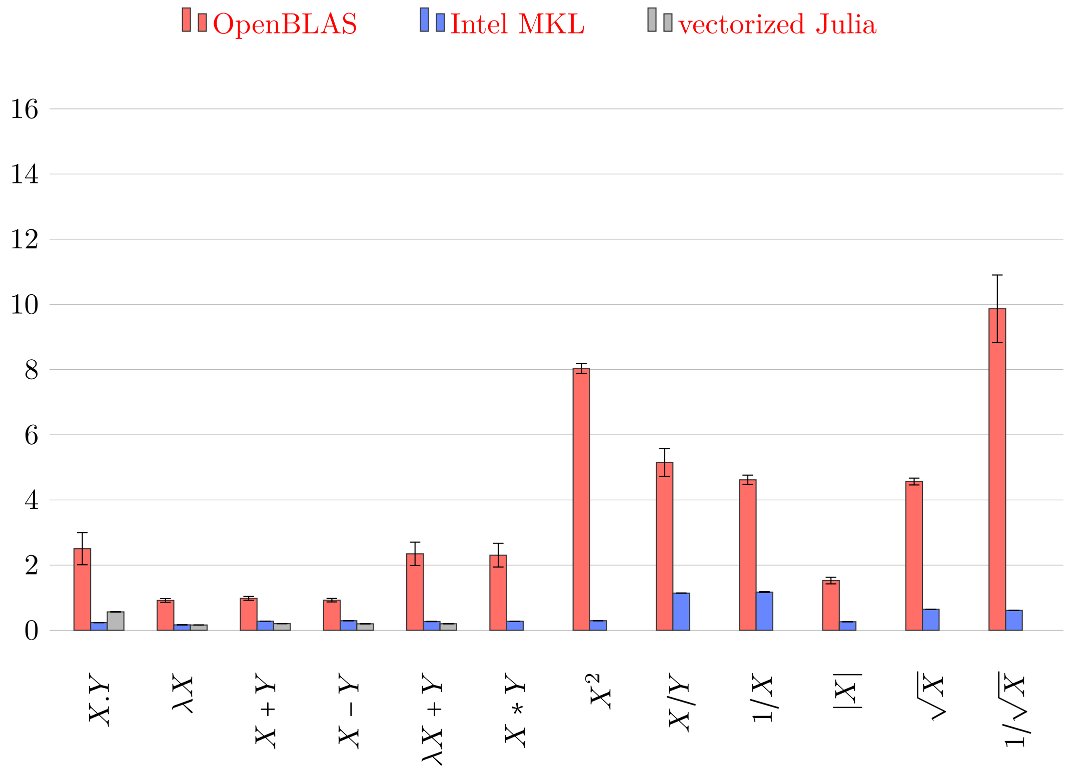
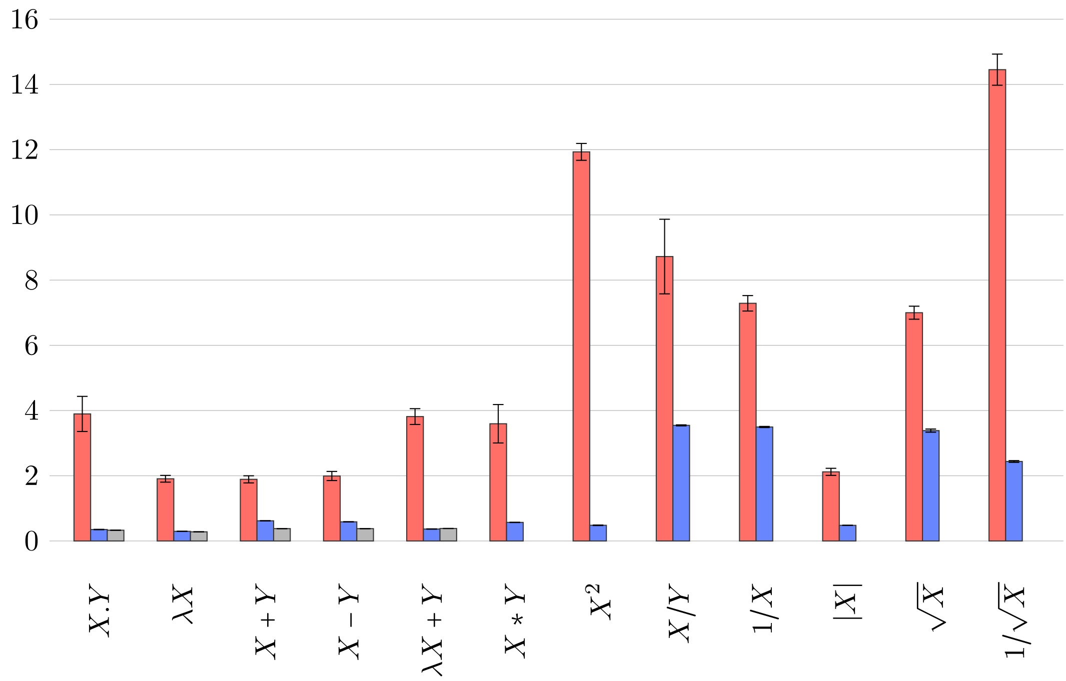


Figure 6.1 – Nonlinear cost of CPU time in ns/el of memory allocation for arrays (Float64).



(a) Float32



(b) Float64

Figure 6.2 – Absolute CPU time in $ns/element$ for $n = 104$ elements. Error bars indicate 95% condifdence interval.

(a) Float32

(b) Float64

Figure 6.3 – CPU time relative to Intel MKL for $n = 104$ elements.

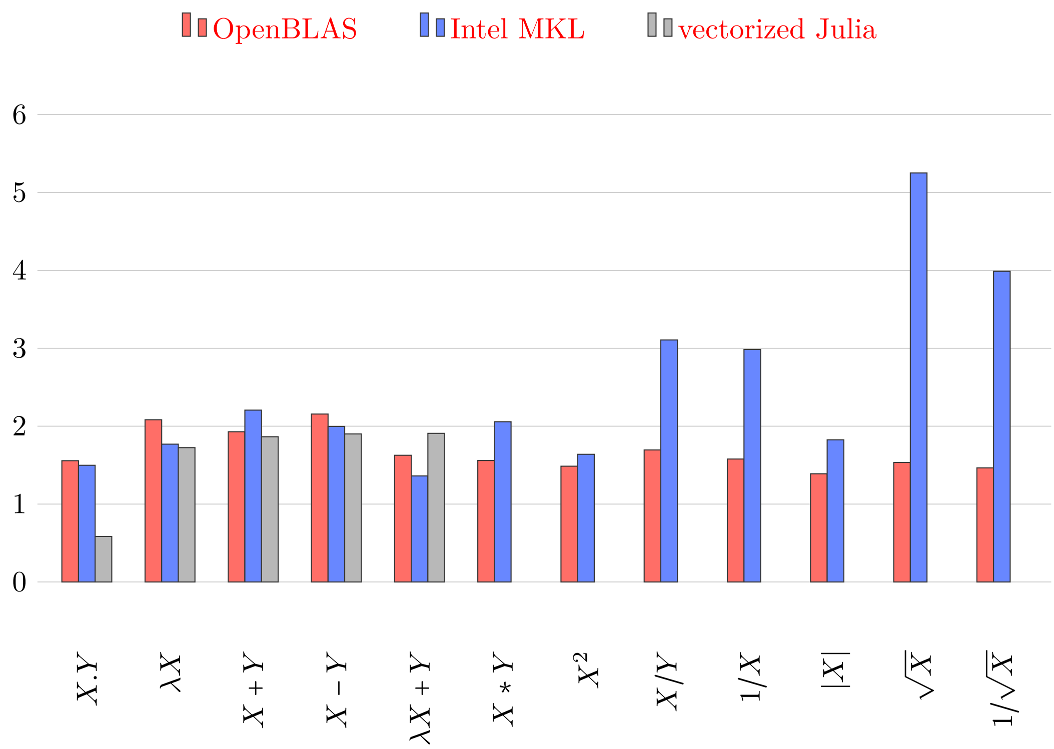


Figure 6.4 – relative CPU time performance for double versus single precision numbers for $n = 10^4$ elements.

profiling : https://software.intel.com/en-us/intel-vtune-amplifier-xe

SIMD : http://www.drdobbs.com/architecture-and-design/simd-enabled-vector-types-with-c/240168888

https://software.intel.com/en-us/articles/optimize-for-intel-avx-using-intel-math-kernel-librarys-basic-linear-algebra-subprograms-blas-with-dgemm-routine

- https://msdn.microsoft.com/en-us/library/ms973852.aspx
- http://www.sebastiansylvan.com/post/why-most-high-level-languages-are-slow/
- http://creamysoft.blogspot.fr/2013/05/c-vs-c-performance.html
- http://www.codeproject.com/Articles/212856/Head-to-head-benchmark-Csharp-vs-NET
- https://software.intel.com/en-us/articles/speeding-up-c-code-with-the-vtune-amplifier-xe-performance-profiler
- http://jonathankinlay.com/index.php/2015/02/comparison-programming-languages/

Bibliography

[Aka12] Eisha Akanksha. Modern CPU ' s memory architecture. *Intelligent Systems and Applications*, 4(April):26–32, 2012.

[Dre07] Ulrich Drepper. What every programmer should know about memory. *Red Hat Magazine*, 2007.

```
using VML

# wrap functions to avoid global scoping while testing
function sqrt_jvectorized{T<:Number}(a::Vector{T})
    sqrt(a)
end
function sqrt_jbroadcast{T<:Number}(a::Vector{T})
    broadcast(sqrt,a)
end
function sqrt_jbroadcast!{T<:Number}(dest::Vector{T}, a::Vector{T})
    broadcast!(sqrt,dest,a)
end
function sqrt_jmap{T<:Number}(a::Vector{T})
    map!(sqrt,a)
end
function sqrt_jmap!{T<:Number}(dest::Vector{T}, a::Vector{T})
    map!(sqrt,dest,a)
end
function sqrt_jloop{T<:Number}(dest::Vector{T},a::Vector{T})
    @inbounds for i in eachindex(a) dest[i]=sqrt(a[i]) end
end

function sqrt_bench()
    # define vector size and floating precision
    n = 1_000
    T = Float64
    # allocate vectors
    dest  = ones(T,n)
    @time a = rand(T,n)
    # bench
    gc()
    gc_enable(false)
    @time sqrt_jvectorized(a)
    @time sqrt_jbroadcast(a)
    @time sqrt_jbroadcast!(dest,a)
    @time sqrt_jmap(a)
    @time sqrt_jmap!(dest,a)
    @time VML.sqrt(a)
    @time VML.sqrt!(dest,a)
    gc_enable(true)
end

sqrt_bench()
```

Listing 1 – Example from external file

```
using DataFrames, VML

function sqrt_bench()

    # vector size
    N = [1,2,3,4,5,6,7,8,9,
         10,20,30,40,50,60,70,80,90,100,200,300,400,500,750,
         1_000,2_500,5_000,7_500,10_000,100_000,1_000_000]

    # dataframe for results
    df = DataFrame(N=[],ALLOC=Float64[],JULIA=Float64[],MKL=Float64[])

    @inbounds for i in 1:length(N)
        T = Float64
        n = N[i]
        a = rand(T,n)
        dest = zeros(T,n)

        # evaluate sqrt and allocation
        # for small n @elapsed applies to a bunch of evaulations
        nrep = 1000
        ncycle = 10_000 ÷ n + 1

        # trigger garbage collection
        gc()
        talloc = 0.0 ; talloc = 0.0 ; tsqrt = 0.0
        for j in 1:nrep
            talloc += @elapsed for k in 1:ncycle Vector{T}(n) end
            tsqrt += @elapsed for k in 1:ncycle sqrt(a) end
        end

        # scale results (ns/element)
        talloc = talloc / nrep / ncycle / n * 1e9
        tsqrt  = tcpu / nrep / ncycle / n * 1e9

        # write results
        push!(df,[n,talloc, tsqrt])
    end
    df
end

sqrt_bench()
```

Listing 2 – Example from external file