PARALLEL PROCESSING

March 14, 2024

UNIVERSIDAD
POLITÉCNICA
DE MADRID

Contents

Abstract

  - Vectorización fundamental a nivel de un solo core.
- Importancia del caché frente a tener que sacar los datos de la RAM (latencia).
- Importancia fundamental de estrategias para optimizar el código, por ejemplo Loop array ordering (column major ordering en Fortran frente a row major orden en C). Inicialmente el código mymatmul que hicimos funcionaba muy mal porque no estábamos teniendo en cuenta que Fortran es major-column order. **Pendiente en punto 3.1**
- Nociones de core vs threads.
- Nociones de microoperaciones, superpipelining, pipelining, etc.
- Esencial la estimación teórica del tiempo que necesitamos para resolver el cálculo (al menos en CPU).
- Instrucciones utilizadas hoy en día como AVX512 o Fused Multiply and Accumulate (FMA) **Marcado en theoretical time (2.4)**
- Block operations para la caché - blocking/slicing/dicing.
- Principales librerías: Intel MKL, BLIS, AOCL. Otros nombres importantes: BLAS, GoTo BLAS, OpenBLAS.
- GPUs trabajan en simple precisión (fundamental cuando comparemos CPU vs GPU).
- Objetivos finales: Qué librería es la que mejor trabaja? Podemos imitar la velocidad de MKL con código en Fortran? (No pudimos, saltamos a usar MKL directamente).
- En Fortran (al menos) es fundamental la configuración que tenemos en el compilador para conseguir las velocidades teóricas que calculamos.
- Comportamiento de las curvas de FLOPS vs tamaño de matriz, había 3 comportamientos: origen, centro y comportamiento en el infinito. **Pendiente en punto 3.1**
- Objetivo final: CPU Multicore vs GPU. **En la introducción**

1 Introduction

2 Understanding the CPU

2.1 Vectorization

2.1.1 The concept of vectorization

Vectorization is a fundamental feature inherent to modern processor architectures. It pertains to the execution of operations on extensive data sets, in contrast to singular data elements. Processors equipped with 512-bit registers can process multiple data elements concurrently. This capability allows for optimal use of the processor's computational resources, enhancing the speed of operations. As a result, operations previously restricted to serial execution can now undergo parallel processing, increasing computational efficiency and reducing execution time for data-intensive tasks.

2.1.2 512-bit vector register

Modern CPUs have vector registers that allow for Single Instruction, Multiple Data (SIMD) operations. The 512-bit wide vector registers allow the CPU to perform arithmetic on multiple data points simultaneously, which can significantly speed up operations, especially in the context of matrix computations.

2.1.3 Thread vs Core "Parallelism"

Threads are the smallest unit of processing that can be scheduled by an operating system. Core- level parallelism, on the other hand, refers to executing multiple tasks simultaneously on different physical cores of a processor.Using threads is useful for hiding latency and is good for I/O bound tasks. Core-level parallelism is more suited for CPU-bound tasks, especially if the threads are not sharing data, as this can lead to a nearly linear speed-up.
Pending to elaborate further on threads and cores: how a thread gets 'vectorized' and how a core gets parallelized (is it intrinsically parallelizable in a core?)

2.2 Memory accesses

2.2.1 RAM speed, latency, and system cache

Different RAM configurations result in different data access times, which could substantially influence the performance of large-scale computations. Generally, operations that involve a high volume of data access can be impacted by RAM speed and latency. Data needs to be fetched frequently from the RAM. If the RAM cannot supply data quickly enough, the CPU might have to wait for the data, leading to a memory bottleneck and a decrease in the overall performance. In this context, **the system cache** is also a key factor. The cache is a smaller, faster type of memory that stores copies of data from frequently used main memory locations. When the CPU needs data, it first looks for it in the cache. If the data is found there (a cache hit), it can be accessed much faster than from RAM. However, if the data is not in the cache (a cache miss), the CPU has to fetch it from RAM, which takes longer.The efficiency of cache usage depends on many factors. One of them is data locality: if the data that needs to be accessed is stored close together, it's more likely to be loaded into the cache at the same time, increasing the chances of cache hits in the future. Additionally, variables store and manipulate data. Static variables have fixed size and memory locations, while dynamic variables, allocated at runtime, pose optimization challenges. The compiler lacks memory layout and size details until runtime, hindering optimization. Thus, static variables are preferred for predictability, aiding code optimization.

2.2.2 About stack and heap:

Optimizing memory usage in a CPU involves strategic allocation of data between the **stack** and the **heap**. The stack, a region of memory dedicated to function calls and local variables, offers fast access but has limited size and fixed allocation. In contrast, the heap provides dynamic memory allocation, accommodating larger and flexible data structures, albeit with slower access. Deciding where to store data impacts performance; leveraging the stack for smaller, short-lived variables reduces memory overhead and improves cache coherence.

Conversely, employing the heap for larger, persistent data ensures flexibility but may lead to increased memory fragmentation and slower access times. Effective optimization necessitates a balance between stack and heap usage, tailored to the application's memory requirements and execution characteristics.

2.3 Micro-operations and pipelining

Micro-operations Pedro: Micro-operations illustrate a processor's capacity to process multiple tasks within one cycle, enabled by pipelining and throughput techniques. In the past, first-generation CPUs typically executed fewer than one operation per cycle. In contrast, contemporary pipelined superscalar processors can perform several operations in a single cycle, although this is constrained by specific limits determined by different CPU processing stages. The micro-operations performance of various processors is tabulated below:

Rong: Although not well understood, a processor can also perform multiple instructions in a single clock-cycle. This concept is related to pipelining and throughput. To simplify, this new factor involved in the theoretical limit can take values ranging from 4 to 8.

2.4 Theoretical time

The discussion will be about the **"theoretical time of a CPU"**, which refers to the time estimated or calculated for a CPU to complete a task under ideal conditions, without considering physical limitations such as memory latency, system bottlenecks, or effects of parallelism in execution. It's a theoretical measure used to evaluate the potential performance of a CPU under various scenarios and workloads.

Escribir aquí sobre FMA, para que tenga sentido el cálculo del tiempo teórico

Based on the above factors, CPU performance can be estimated as:

$$t_{CPU} = \frac{N_{ops} \times 4}{V_{vectorization} \times GH_{z_{CPU}} \times M_{micro-ops} \times C_{CPU}}$$

Where the parameters are:

1. $V_{vectorization}$: Vectorization factor: 16 (512-bit)
2. $M_{micro-ops}$: Micro-operations factor: 4, 6 (AMD Zen 3) or even 8 (Apple Silicon)
3. $GH_{z_{CPU}}$: Clock speed of the CPU
4. C_{CPU}: Number of cores in the CPU
5. $\times 4$: accounts for the 4-step sequence in matrix multiplication, as discussed earlier.

3 Optimization: approaching theoretical operation of the CPU

3.1 Matrix operations

GEMMs(GeneralMatrixMultiplications) Math And Memory Bounds
A Layered Approach to GEMM

Cache Blocking To efficiently utilize the cache, we implemented cache blocking in matrix multiplication. The idea is to divide the matrix into smaller blocks that fit into the cache, reducing the number of cache misses. However, it is important to note that cache blocking alone still performs the matrix multiplication sequentially. This means that it only uses a single core of the CPU. In fact, using a single-threaded approach with cache blocking can sometimes result in worse performance compared to a simple matrix multiplication due to the additional loops and steps involved in handling the blocks. The- refore, while cache blocking is an essential optimization technique for reducing cache misses, the performance might still be suboptimal if the computation is not parallelized to take advantage of multiple cores.

3.1.1 Column major ordering in Fortran vs row major ordering in C

3.1.2 Comportamiento de las curvas de FLOPS vs tamaño de matriz

C++ Benchmarks

The C++ benchmarks compared single-core and multi-core performance using an Intel i5-10210U processor. Initially, single-core execution took several minutes for 10,000 iterations, with CPU usage peaking at around 90% but often hovering around 60%. Utilizing OpenMP in Visual Studio C++, parallel processing significantly reduced execution time as all cores reached 100% utilization when connected to a power supply, stabilizing at 80% when on battery power. The results showed that multi-core performance outperformed single-core, with OpenMP achieving a time of 102.795 seconds compared to 565.581 seconds for single-core.

Additional benchmarks were conducted in Fortran, MATLAB, and Python. Fortran, using either the matmul function or a simple loop, did not reach full CPU utilization but achieved similar speed to OpenMP in C++. MATLAB and Fortran achieved results similar to C++ despite not utilizing 100% of the CPU. Python, although not reaching full CPU utilization, provided the best results, potentially due to MATLAB limiting CPU usage.

In conclusion, C++ with OpenMP outperformed other languages in terms of utilizing CPU cores fully, while Fortran and MATLAB achieved similar performance despite not reaching full CPU utilization. Python yielded the best results, possibly due to CPU usage limitations in MATLAB. Further understanding of BLAS, especially in Python and MATLAB, could provide insights into optimizing performance.

Analysis of Theoretical Limits

The analysis focuses on determining whether an expensive processor or a GPU cluster offers better performance within a given budget. A model is proposed to predict the maximum speed achievable by a GPU cluster.

Theoretical Time

The theoretical time is estimated using the equation:

$$t^* = \frac{4 \times \text{Number of operations}}{\text{Clock Speed} \times \text{Vectorization} \times \text{Number of threads}}$$

Where the number of operations is equal to $2 \times N \times M \times \text{TIMES}$, the vectorization argument is $\frac{512}{32}$ (using single precision), and the number of threads is 1. For reference, with an Intel i5-10210U processor clocked at 2.1 GHz, a matrix of size 5000×5000, and 1000 iterations, the theoretical time is calculated as 5.952 seconds, though this isn't entirely accurate due to the processor not supporting AVX-512 extension.

Column and Row Operations

Comparing performance, it's noted that a conventional matrix-vector multiplication method is inefficient for the processor due to memory access inefficiencies. A revised function operating by columns significantly improves performance.

Using the new method, the time decreased from 343.264 seconds to 28.637 seconds, achieving a speedup of nearly 12. No optimization options were applied during testing, as activating them would result in similar times due to vectorization.

Overall, the analysis demonstrates the significance of efficient memory access and algorithmic optimization for enhancing performance, especially in matrix operations.

Size Influence

If a matrix fits within the CPU cache, processing is faster; otherwise, accessing memory outside the cache slows down processing. The method involves conducting tests with various matrix dimensions and iterations to maintain constant operations.

Cache and Pipelining

Block sections

By partitioning the matrix and resulting vector into blocks, the implementation in Fortran efficiently utilizes cache memory, thereby reducing the number of multiplications and maximizing speed.

BLAS Optimization

Activating BLAS may utilize all cores and may not always yield the expected results. BLAS is designed to optimize linear algebra operations (matrix multiplication etc), and it is still widely used as of today due to its performance and efficiency. Now, in the context of Fortan, and compilers such as GFortran and Intel Fortran Compiler(ifort) you have an option to use BLAS to optimize your operations, -fexternal-blas in the case of GFortran for example, by doing this you are telling the compiler to use the BLAS implementations available. Theoretically this should make the compiler use all the CPUs thus making the performance increasingly better. This, however, does not always happen, yielding unexpected results. This can be due to the code structure, or the workload distribution around the cores, though this are only hipothesis.Overall,using BLAS in theory you should have better results but this has not been the truth in practice.