Impact of vectorization and multithreading on performance and energy consumption on Jetson boards

Sylvain Jubertie, Emmanuel Melin, Naly Raliravaka, Emmanuel Bodèle, Pablo Escot Bocanegra

Impact of vectorization and multithreading on performance and energy consumption on Jetson boards

Sylvain Jubertie
LIFO EA 4022
Université d'Orléans, INSA CVL
Orléans, France
sylvain.jubertie@univ-orleans.fr

Emmanuel Melin
LIFO EA 4022
Université d'Orléans, INSA CVL
Orléans, France
emmanuel.melin@univ-orleans.fr

Naly Raliravaka
LIFO EA 4022
Université d'Orléans, INSA CVL
Orléans, France
rali@univ-orleans.fr

Emmanuel Bodèle
IUT Département GTE
Université d'Orléans
Orléans, France
emmanuel.bodele@univ-orleans.fr

Pablo Escot Bocanegra
GREMI UMR 7344
CNRS/Université d'Orléans
Orléans, France
pablo.escot@univ-orleans.fr

***Abstract*—ARM processors are well known for their energy efficiency and are consequently widely used in embedded platforms. Like other processor architectures, they are built with different levels of parallelism, from Instruction Level Parallelism (out-of-order and superscalar capabilities) to Thread Level Parallelism (multicore), to increase their performance levels. These processors are now also targeting the HPC domain and will equip the Fujitsu Post-K supercomputer. Some ARM processors from the Cortex-A series, which equip smartphones and tablets, also provide Data Level Parallelism through SIMD units called NEON. These units are able to process 128-bit of data at a time, for example four 32-bit floating point values. Taking advantage of these units requires code vectorization which may be performed automatically by the compiler or explicitly by using NEON intrinsics. Exploiting all these levels of parallelism may lead to better performance as well as a higher energy consumption. This is not an issue in the HPC domain where application development is driven by finding the best performance. However, developing for embedded applications is driven by finding the best trade-off between energy consumption and performance.**

In this paper, we propose to study the impact of vectorization and multithreading on both performance and energy consumption on some Nvidia Jetson boards. Results show that depending on the algorithm and on its implementation, vectorization may bring a similar speedup as an OpenMP scalar implementation but with a lower energy consumption. However, combining vectorization and multithreading may lead close to both the best performance level and the lowest energy consumption but not when running cores at their maximum frequencies.

Index Terms*—vectorization, SIMD, multithreading, energy consumption

I. Introduction

The ARM architecture has evolved in the same way as the x86 architecture by adopting parallel technologies such as SIMD (Single Instruction on Multiple Data) units and multicore designs to increase the performance of processors. Furthermore, multicore architectures are also interesting in that they decrease the energy consumption of a chip by disabling some of its cores and reducing their frequencies. The ARM big.LITTLE architecture also offers a combination of different clusters of cores, some with faster power-hungry cores and some with slower energy-efficient cores, within the same processor. Depending on the workload, processes may be migrated from one cluster to another.

SIMD units provide another way to increase the core performance by applying a single instruction to several data at the same time depending on its width. The ARM SIMD unit is called NEON and is 128-bit wide. Thus, it is able to process four 32-bit floating point values or sixteen 8-bit integral values at the same time so we can theoretically expect the corresponding performance speedup. Since processing data in a SIMD way requires fewer instructions to decode (one SIMD instruction replaces several scalar ones), we can expect a slightly lower energy consumption than for a scalar computation. However, SIMD units are larger than scalar ones thus requiring more power to perform one SIMD instruction compared to its scalar version. On x86 processors, using larger AVX and AVX-512 units automatically reduces the core frequency to limit the power drained by the cores. This is not the case for smaller NEON units. Thus, using SIMD units may reduce the computation time thus the energy needed for the computation, while at the same time it increases the power required by the cores. Several approaches are available for code vectorization. First, compilers may be able to perform automatic vectorization in some cases. However, there is no guarantee that the vectorization occurs or that the resulting vectorized code is optimal. Adding OpenMP SIMD pragmas in front of vectorizable loops may help the compiler vectorization process. Another possibility is to use

the compiler SIMD intrinsics which offers specific types and instructions to explicitly vectorize the code. Since intrinsics are not portable from one architecture to another, one can also use libraries such as Boost.SIMD[1]/bSIMD or MIPP[2], written in C++, which allow higher level abstractions, by providing vector types and overloaded operators. It is also possible to use domain-specific SIMD optimized libraries such as MKL (only for x86 platforms), Eigen, OpenBLAS, FFTW or Arch-R.

In order to determine if exploiting SIMD units and multithreading is interesting for both performance and energy consumption, we propose to run different codes on some Nvidia Jetson boards equipped with ARM processors. We measure the energy by means of a measurement platform inserted between the power supply and the board. For each code, we measure the required amount of energy and the speedup provided by its explicit vectorized version over its reference scalar version. Theoretically, a NEON unit in a single core may offer the same speedup as four cores in scalar mode when processing 32-bit floating point values. Thus, an OpenMP version is also provided to compare the effects of vectorization versus multi-threading and also to study the combination of both approaches.

The paper is organized as follows. In section 2, we present the related work. In section 3, we give more details about boards characteristics and the measurement platform. In section 4, we present the algorithms used for our experiments and their different implementations. Then, in section 5, we discuss the experimental results. Finally, we conclude on our current work and propose some future studies.

II. Related work

The energy-performance trade-off may be studied from the architecture side, by modifying hardware characteristics, from the software side, by studying code optimizations.

On the architecture side, many papers investigate the use of Dynamic Voltage and Frequency Scaling (DVFS) on various architectures. In [3], the use of DVFS is driven by a predictive model to guarantee some QoS criteria. This study focuses on the activation of x86 CPU cores and the modification of their frequencies depending on the predicted workload to optimize both the throughput and the latency. In [4] and [5], the authors study the impact of DVFS on the energy-performance trade-off for an HPC Lattice Boltzmann application on two different architectures: respectively an ARM-based Jetson TK1 board and an x86 platform with dedicated GPUs. Results show that it is an interesting approach for memory-bound codes.

On the software side, several approaches are investigated. In [6], the authors propose a compilation framework based on an energy model to study the influence of SIMD operations on energy consumption for a Digital Signal Processor (DSP). On this particular architecture, applying this approach to several algorithms allows an average reduction of 72% of the energy needed and of 76% of the execution time over a scalar implementation. GPUs are also composed of big SIMD units. In [7], the authors measure the impact of some source code optimizations on performance and energy consumption. In [8], the authors propose a measurement technique on GPUs to estimate the per-component power of each GPU component, thus providing a model for estimating the energy consumption. At a higher level, the impact of the parallel programming paradigm is also considered in [9][10].

In this paper, we propose to combine both architecture and software sides to optimize the performance and energy consumption on ARM-based platforms. On the architecture side, DVFS techniques allow us to enable cores, to change their frequencies and also to modify the memory frequency. On the software side, we study the effect of both multithreading and vectorization.

III. Experimental platform

A. Jetson boards

We consider the Jetson TK1 and TX1 boards, running their default Ubuntu operating system. We chose these boards as they are equipped with similar ARM-based processors as those widely used in mobile devices such as smartphones and tablets. Boards characteristics are summarized in table 1. The TK1 is equipped with five 32-bit ARM cores divided into two clusters. The first cluster is called Low Power (LP) and contains a single energy efficient core which is mainly used when the system is in idle state. The second one contains four Cortex-A15 cores and is used for more complex tasks. The TX1 is more recent and is equipped with four 64-bit Cortex-A57 ARM cores. These boards are also equipped with embedded GPUs which share the same memory, but we only consider the CPU cores in our study. It is possible to manually control the CPU and memory frequencies and the number of active cores through a simple interface provided by the Linux kernel. This is an important feature which justifies the use of these ARM platforms for our study since x86 platforms do not provide such a fine control on cores and memory.

	Jetson TK1	**Jetson TX1**
processor	4x Cortex-A15	4x Cortex-A57
cpu freq.	51 - 2,065	102 - 1,734
cache	32KB L1D/128KB L2	32KB L1D/2MB L2
memory	2GB DDR3L	4GB LPDDR4
mem. freq.	12.75 - 924.0	40.8 - 1,600.0
GPU	192 Kepler cores	256 Maxwell cores
GPU freq.	72.0 - 852.0	76.8 - 998.4

Fig. 1: Jetson boards characteristics (freq. in MHz).

B. Measurement platform

Our measurement platform is composed of 2 stabilized power supplies, a Jetson board, an Arduino board, a current sensor and a PC computer to collect data (see figure 2). A stabilized power supply provides a good quality 19V current to the Jetson board. A hall effect sensor measures this current intensity. To stabilize our measurements we choose not to take into account the fan current. Fans are connected to a separate power supply. The Arduino converts the analog output of the sensor into digital information at 500Hz frequency. The Arduino stores this information and uses a USB connection

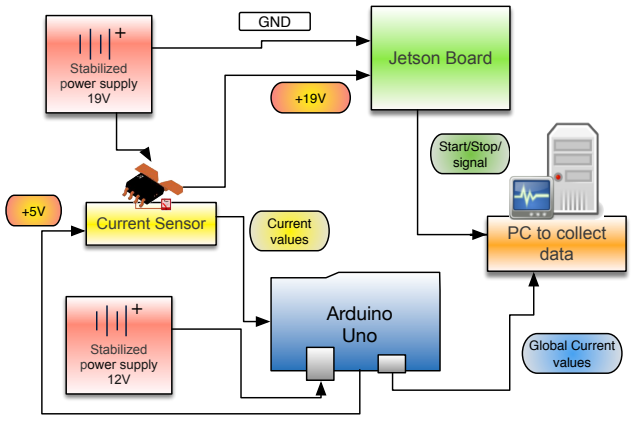

Fig. 2: Measurement platform

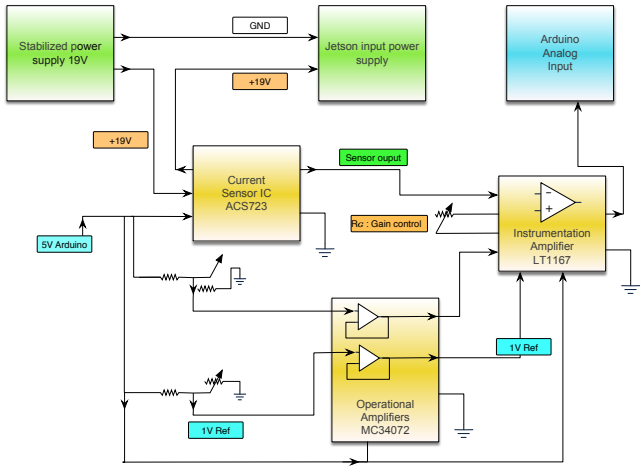

Fig. 3: Converter

to send an average value, on a regular basis (every 0.1s), to a PC computer which gathers all the data. The Jetson uses a serial connection to send a signal to the PC informing it of the beginning and the end of a run. This allows the PC to know the duration T (second) of a run and the average intensity I_i (Ampere) for each time unit t_i (second). Note that all time units are equal to $t = 0.1s$. The total number of time units N during a run is given by $N = T/t$, we have: $T = \sum_{i=1}^{N} t_i$. For each time unit we deduce the average power $P_i = I_i * 19volt$ (Watt). The total energy consumption of a run is given by $E = \sum_{i=1}^{N} t_i * P_i$ (joule).

The core of our measurement platform is composed of a customized analog converter which measures the current applied to the board. To guarantee measurement accuracy, we took special precautions with the design of this instrument. The converter components are described in figure 3. It works as follows. A +19V stabilized power goes through a hall effect current sensor IC (ACS723) to supply the Jetson TX1 (resp. +12V for the TK1). This sensor generates a proportional voltage with a sensibility of 0.4mV/A. The zero current output voltage is half the supply voltage, here about $V_{cc} = 5V$ depending on the USB powering the Arduino. So we just need to compute $I(A) = \frac{(V_{outsensor} - \frac{5}{2})}{0.4}$ to obtain the current intensity. Note that we need to transform the ASC723 output signal in order to exploit the whole range of the Arduino analog input (i.e. between 0V and 5V), so we use an instrumentation differential amplifier (LT1167) between the sensor and the Arduino. This IC amplifies by a factor G the voltage difference between the sensor output (V_{in^+}) and an adjustable V_{in^-}= $\frac{5}{2}$V (which corresponds to the zero current output of the ACS723) and adds this result to a reference voltage V_{ref} (an adjustable 1V). The Arduino analog input voltage V_{out} is then given by : $V_{out} - V_{ref} = G(V_{in^+} - V_{in^-})$ this gives here: $V_{out} - 1 = G(V_{outsensor} - \frac{5}{2})$ and then: $V_{outsensor} = \frac{(V_{out}-1)}{G} + \frac{5}{2}$. The gain is computed from this equation: $G = \frac{49.4k\Omega}{R_G} + 1$. The two references $\frac{5}{2}$V and 1V should not be altered, otherwise, measurements may be less accurate. To ensure this, we isolate them with operational amplifiers used in the circuit as follows (MC34072). Finally, we obtain the current consumed by the Jetson with this equation:

$$I(A) = \frac{(\frac{(V_{out}-1)}{G})}{0.4} = \frac{(\frac{(V_{out}-1)}{\frac{49.4k\Omega}{R_G}+1})}{0.4} = \frac{(\frac{(\frac{DigitalOutput_{Arduino}}{1023}V_{cc}-1)}{\frac{49.4k\Omega}{R_G}+1})}{0.4}$$

To sum up, the use of this custom design offers the capability to adjust and isolate reference voltage and to exploit the whole range of the Arduino analog input. This is the main contribution of this converter.

IV. Algorithms

We consider two different algorithms in this study which are good candidates for vectorization. Other algorithms were also studied like an YUV to RGB image conversion and vector normalization but are not presented here since results are similar to those obtained with the following grayscale image conversion.

A. Grayscale image conversion

We have chosen this algorithm since, in the image processing domain, it is often the first step performed before applying more complex filters, for example edge detection filters like the Sobel one. It is a simple algorithm which, for each pixel of an image, consists in taking its three components and computing the resulting level of gray according to the following formula:

$$gray = \frac{307 * R + 604 * G + 113 * B}{1024} \quad (1)$$

Components are encoded as 8-bit unsigned integers and are stored in an Array of Structure (AoS) layout. These values are accumulated, thus it is necessary to use 32-bit unsigned

integers for intermediate results. The scalar implementation is straightforward. However the NEON implementation is much more complex and requires more than fifty lines of code. Listing 1 shows a condensed version of this code to illustrate the verbosity brought by the use of intrinsics. The `vldq3q_u8` intrinsics loads 48 8-bit values, i.e. 16 pixels, and deinterleaves them into a structure composed of three separate NEON registers, one for each color component. Most additional intrinsics are used to convert 8-bit values to 32-bit, then back to 8-bit. Note that, `vget_low` and `vget_high` intrinsics do not produce assembly instructions since 128-bit NEON registers are composed of two 64-bit registers directly accessible with their own register names.

Sixteen pixels are loaded at the same time, but computation is done four by four pixels since 8-bit values need to be converted to 32-bit values. In this case, we can expect computation to be the preponderant limiting factor over memory transfers and thus giving a speedup close to the theoretical one when operating on 32-bit values. We can expect a multithreaded version to have a similar behavior.

Listing 1: Grayscale NEON version.

```
for(std::size_t i=0 ; i < out.size()/16*16 ; i+=16) {
  // Load 16 pixels (AoS -> SoA).
  uint8x16x3_t rgb = vld3q_u8( &in[ 3 * i ] );
  // Split 8-bit vectors.
  uint8x8_t r0 = vget_low_u8( rgb.val[ 0 ] );
  uint8x8_t r1 = vget_high_u8( rgb.val[ 0 ] );
  uint8x8_t g0 = vget_low_u8( rgb.val[ 1 ] );
  uint8x8_t g1 = vget_high_u8( rgb.val[ 1 ] );
  uint8x8_t b0 = vget_low_u8( rgb.val[ 2 ] );
  uint8x8_t b1 = vget_high_u8( rgb.val[ 2 ] );
  // Convert uint8 to uint16.
  uint16x8_t R0 = vmovl_u8( r0 );
  uint16x8_t R1 = vmovl_u8( r1 );
  uint16x8_t G0 = vmovl_u8( g0 );
  uint16x8_t G1 = vmovl_u8( g1 );
  uint16x8_t B0 = vmovl_u8( b0 );
  uint16x8_t B1 = vmovl_u8( b1 );
  // Split 16-bits vectors.
  uint16x4_t R00 = vget_low_u16( R0 );
  uint16x4_t R01 = vget_high_u16( R0 );
  uint16x4_t R10 = vget_low_u16( R1 );
  uint16x4_t R11 = vget_high_u16( R1 );
  // Multiply uint16 vectors, convert to uint32.
  uint32x4_t O00 = vmull_u16( R00, v307 );
  uint32x4_t O01 = vmull_u16( R01, v307 );
  uint32x4_t O10 = vmull_u16( R10, v307 );
  uint32x4_t O11 = vmull_u16( R11, v307 );

  // Same as 8 lines above for the G and B components.

  // Shift by 10 bits and convert back to 16-bit.
  uint16x4_t P00 = vshrn_n_u32( O00, 10 );
  uint16x4_t P01 = vshrn_n_u32( O01, 10 );
  uint16x4_t P10 = vshrn_n_u32( O10, 10 );
  uint16x4_t P11 = vshrn_n_u32( O11, 10 );
  // Merge vectors.
  uint16x8_t T0 = vcombine_u16( P00, P01 );
  uint16x8_t T1 = vcombine_u16( P10, P11 );
  // Convert back to uint8 values.
  uint8x8_t U0 = vmovn_u16( T0 );
  uint8x8_t U1 = vmovn_u16( T1 );
  // Merge into a single register.
  uint8x16_t result = vcombine_u8( U0, U1 );
  // Store register to memory.
  vst1q_u8( &out[ i ], result );
}
```

B. Matrix multiplication

Multiplying two matrices A and B basically consists in performing the dot product of a line of A with a column of B for each element in the resulting matrix C. The resulting code consists of three nested loops, the two outer ones to traverse matrix C and the innermost one to perform the dot product. However, this approach is not cache efficient nor SIMD friendly since elements in matrix B are not accessed contiguously. A classic way to solve this is to swap the two innermost loops. The NEON implementation is straightforward, it only requires modifying the inner loop to process four 32-bit floating point values at a time.

The performance behavior of this algorithm may be different from the previous one since only a Fused Multiply-Add instruction is performed by iteration while requiring two register loads and one store. For large matrices, which do not fit into cache, performance may be limited by the memory bandwidth. Thus, we can study the behavior of the combination of both cache and NEON mechanisms and compare it with the multithreaded version.

In the experimental results section, we also compare the behavior of our implementation to those obtained with the highly optimized OpenBLAS library. This library contains specific ARM AArch32 and AArch64 NEON implementations written in assembly and performs matrix transformation prior to the computation to optimize memory accesses. Thus, we are also able to discuss the impact of memory optimization on performance and energy consumption.

Listing 2: Matrix multiplication, NEON version.

```
for( std::size_t j = 0 ; j < dim ; ++j ) {
  std::size_t js = j * dim;
  for( std::size_t k = 0 ; k < dim ; ++k ) {
    int ks = k * dim;
    float32x4_t rA = vld1q_dup_f32( &A[ js + k ] );
    for( std::size_t i = 0 ; i < dim ; i+=4 ) {
      float32x4_t rC = vld1q_f32( &C[ js + i ] );
      float32x4_t rB = vld1q_f32( &B[ ks + i ] );
      rC += rA * rB;
      vst1q_f32( &C[ js + i ], rC );
    }
  }
}
```

V. Experimental results

A. Setup and protocol

The GCC 7 and Clang 5 compilers were both used in our experiments. Since Clang offers similar or slightly better results on both platforms we only present results obtained with this compiler. We enable compiler optimizations by adding the `-O3` flag and setting the `-march` flag to the corresponding ARM architecture. For the TK1 board, we also add the flag `-mfpu=neon-vfpv4` to enable the generation of FMA instructions. To prevent automatic vectorization (by default, the `-O3` flag enables the `-ftree-vectorize`, `-vectorize-loops` and `-vectorize-slp` flags), we add the `-fno-tree-vectorize`, `-fno-vectorize` and `-fno-slp-vectorize` compilation flags. Note that, we have also studied the impact of automatic vectorization on

the grayscale and on the matrix multiplication codes. In [11], we show that both GCC and Clang compilers are able to automatically vectorize the grayscale code only on the TX1 board. However, we noticed some very erratic behavior of the compilers on both boards and showed that automatic vectorization is not reliable to obtain a systematic and coherent speedup. Furthermore, for both grayscale and matrix multiplication codes, we always obtain better results by using intrinsics. Thus in this study we only discuss results obtained with scalar and explicitly vectorized versions of the codes.

For each code, we measure its performance and energy consumption for four different versions: 1) scalar 2) NEON 3) scalar+OpenMP with four threads 4) NEON+OpenMP version with four threads. For the grayscale code, we present additional results obtained with two cores to discuss the scalability depending on the number of cores. The scalability of the matrix multiplication code is not presented since we observe a similar behavior. Tests are performed using the userspace governor which allows manual control of both core and memory frequencies as well as enabling or disabling the cores individually. We have purposely ignored the lowest frequencies in our experiments since we have experienced instabilities in the system behavior or in our results.

For the sake of clarity, we only present and discuss the results obtained on the TX1 board since codes scale the same way and similar behavior are observed on the TK1.

B. Results for the grayscale code

Figures 4 and 5 show the execution time and energy consumption for the different implementations of the grayscale code depending on the CPU frequency. Each test consists in performing 30 iterations on an input RGB image of size 14400x7200 (311.04 MB). The memory frequency is fixed to 1.6 GHz which is the highest possible value. We first discuss the execution time of the different implementations.

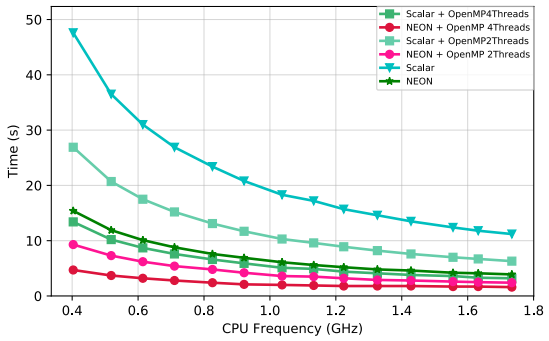

Fig. 4: Grayscale: execution times depending on the CPU frequency.

The scalar code scales very well with the CPU frequency and the number of cores, in this case the code is compute-bound. The NEON and OpenMP implementations scale very well at low frequencies and offer respective speedups of 2.9 and 3.5 with four threads over the scalar version, close to the ideal speedup of 4. At all frequencies, the OpenMP

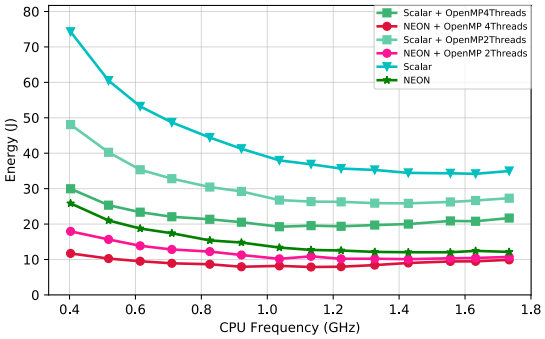

Fig. 5: Grayscale: energy consumption depending on the CPU frequency.

version with four threads is around 15% faster than the NEON version. At 400MHz, the NEON+OpenMP version is around 3 times faster than the OpenMP and NEON versions. However, increasing the CPU frequency does not provide any significant speedup above 1GHz. In this case, the code is likely to become memory-bound.

The energy consumption does not scale in the same way as the execution time. For the scalar version, the energy consumption decreases until the CPU frequency reaches 1.428GHz. Then, it remains the same and rises slightly at 1.734GHz. The curves for the other versions exhibit the same pattern, but not at the same frequencies and with the same range. We observe that OpenMP versions without NEON require much more energy, at least two times more that their corresponding OpenMP+NEON versions. The OpenMP version with four threads reaches its minimum energy consumption at 1.224GHz, the NEON version at 1.428GHz, and the NEON+OpenMP version at 1.036GHz. However, the NEON version consumes less energy than the OpenMP version and the gap between these two versions grows with the CPU frequency.

From figures 4 and 5 we can make the following statements for the grayscale versions with a memory frequency set to 1.6GHz. The NEON version consumes less energy (up to 45% at the highest CPU frequency) than the OpenMP version but is slightly slower (around 15% at all CPU frequencies). The energy consumption does not scale as the performance. This is due to the energy cost of the computation which represents only a fraction of the base energy consumption of the CPU. When the code is compute-bound, increasing the CPU frequency brings more performance while reducing the energy consumption up to some point. Then, performance continues to increase but energy consumption stagnates then starts growing. We explain this by the increase of the CPU voltage required to sustain higher CPU frequencies. Figure 6 shows the voltage applied to the CPU depending on its frequency. Indeed, the average power consumption of a CPU can be modeled as in [5]:

$$P_{avg} = fCV^2 + P_{static} \quad (2)$$

where f is the CPU frequency, C the capacitance of transistor

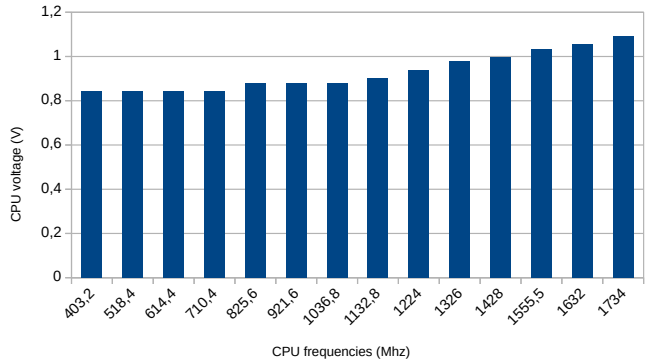

Fig. 6: CPU voltage depending on the frequency.

gates, V the supply voltage and P_{static} the static power when the CPU is idle, also accounting for leakage currents. Thus, if increasing the CPU frequency does not provide enough additional performance, then energy consumption may increase. For the grayscale algorithm, it is possible to achieve best performance while consuming less energy with the NEON+OpenMP version and a CPU set at 1.036GHz.

We perform the same test for each memory frequency available. Figure 7 shows the results in 3D when both memory and CPU frequencies are modified. The color represents the energy consumption for the corresponding core and memory frequencies. We observe that above 800MHz, the memory frequency has very few impact on performance for all the versions. This is not expected since we have stated above that the code is likely to become memory-bound at higher CPU frequencies and core numbers. To explain this behavior, we propose to study how the memory bandwidth evolves with its frequency and the number of cores.

C. Study of the memory bandwidth

To determine the memory bandwidth depending on the External Memory Controller (EMC) frequency we run the STREAM benchmark on both platforms. Figure 8 shows results obtained with the STREAM Copy benchmark on the TX1 platform for different EMC frequencies and for 1, 2 and 4 threads. Results on the TK1 are lower since the memory is one generation older and clocked lower as detailed in table 1.

We observe that the memory bandwidth does not scale linearly with the EMC frequency. Thus, we can not expect a linear speedup when increasing the EMC frequency for memory-bound codes. Another interesting result is that using 4 threads allows us to reach a higher memory bandwidth. However, it does not scale linearly with the number of cores since cores share the same memory controller.

D. Results for the matrix multiplication code

We perform the matrix multiplication as explained in section IV-B. The matrix size is set to 2048×2048. Results are shown in figure 9.

The NEON version is not as fast as expected and provides only a speedup of 1.2 over the scalar version. The OpenMP version on the other hand offers an ideal speedup of 4 at the lowest core frequencies and of 2.7 at the highest ones. In this particular case, the NEON version is not efficient since the OpenMP version offers a better performance and a lower energy consumption. This is due both to a bad cache usage and to not enough interleaving of NEON instructions to hide their latencies. The matrix B does not fit into the L2 cache and the NEON version needs to traverse this matrix for computing each line of matrix C, thus matrix B is loaded 2048 times. The OpenMP version splits the external loop across four threads. Each thread performs the multiplication of a sub-matrix of A by the matrix B. Thus, when one of the threads accesses some data in matrix B for the first time, these data are placed in the L2 cache. As a result, the same data only needs to be sent from L2 to L1 cache when the other threads need to access them. In the best scenario where all threads are synchronous, matrix B is only traversed 512 times.

However, combining OpenMP and NEON for this code is particularly interesting since it provides no performance improvement but reduces the energy consumption by around 20% at all frequencies. Our hypothesis is that the combination of the two following mechanisms may explain this behavior: 1) Adding NEON intrinsics to the OpenMP code does not improve the performance at all, meaning that the code is likely to be memory-bound. In this case, the NEON unit waits four times more than a scalar unit, thus the core may require less energy. 2) Decoding a NEON instruction takes the same amount of time and energy than a scalar unit. Thus, if the NEON instructions are less frequent and cost the same as scalar instructions, the energy consumption automatically decreases. Other mechanisms like power gating may also automatically disable some processor parts in such cases. However processor implementation details at this level are not available.

The OpenBLAS library provides a more optimized implementation by reorganizing matrices, performing computation by tiles, and completely unrolling the computation of each tile to hide NEON instruction latencies. Results obtained with this library show that for one thread it is around 5 times faster than the previous NEON implementation. However, it is difficult to determine which part of the speedup is brought about by the use of NEON or by the cache improvement since no comparable scalar implementation is available. The OpenMP version provides an additional speedup of 3. Note that, in this case we observe the same behavior as with the grayscale algorithm: at some point, performance does not improve by increasing the CPU or memory frequency, but the energy consumption increases.

E. Remarks on the OpenMP vs NEON energy consumption

For the grayscale code, the gap in energy consumption between the NEON version on one core and the OpenMP version on four cores is at most 42% (CPU frequency set at 1.734GHz in figure 5). We would have expected a much greater difference between the NEON version and the OpenMP version, since only one core is running instead of four. To find

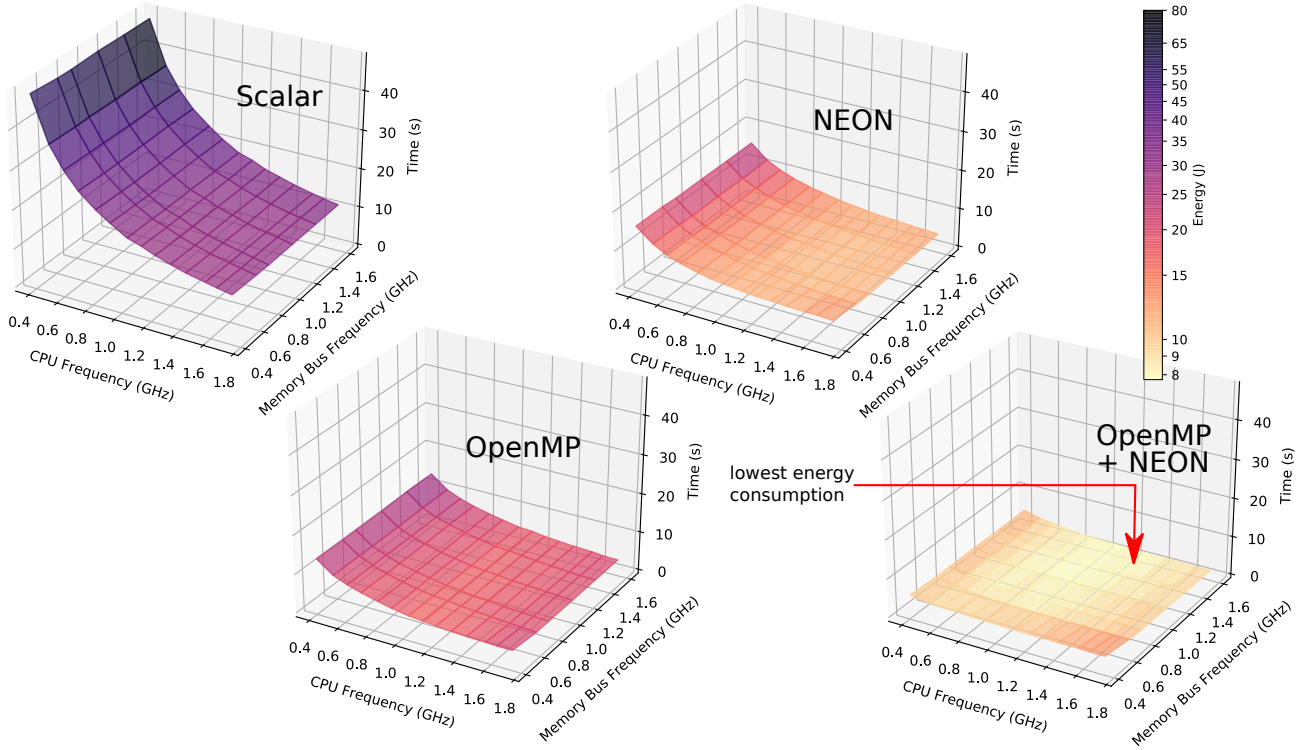

Fig. 7: Performance of the grayscale implementations depending on the CPU and memory bus frequencies. The color represents the energy.

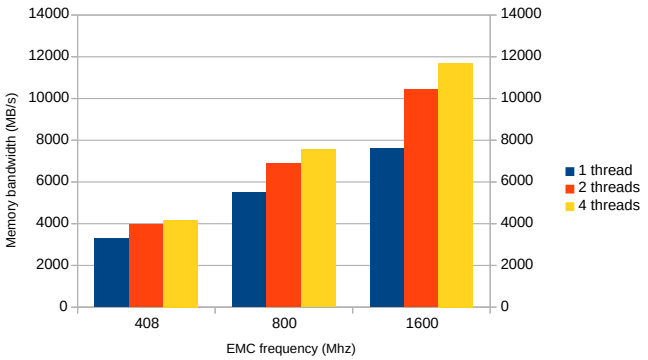

Fig. 8: STREAM Copy results for the TX1 board

an explanation, we have compared the energy consumption obtained for the NEON version (one thread only) with the two following configurations: 1) when using the ondemand governor with a CPU frequency capped at 1.6GHz, 2) when using the userspace governor with all the cores enabled at 1.6GHz. The ondemand governor may disable some cores since the process resides only in one core, thus reducing the energy consumption. However, we obtain the exact same energy consumption in both cases. This suggests that cores are not idle at the hardware level i.e. that P_{static} is the preponderant term in formula 2. Thus, it prevents a significant lower energy consumption for the NEON version over the OpenMP version.

VI. Conclusion and Future Work

We show in this paper that better performance as well as lower energy consumption, in comparison with scalar implementations, are possible when codes are SIMD friendly. In the best case, vectorization brings speedup close to the theoretical one, and similar speedup as multithreading with OpenMP, while requiring less energy. On codes with cache issues, like the matrix multiplication implementation we propose, the vectorization only brings a fraction of the theoretical speedup but still requires less energy than the scalar implementation. In all the cases, the combination of NEON with OpenMP allows to achieve both the best performance and the lowest energy consumption.

Several directions are possible following this work. We plan to extend our study to new ARM architectures, especially big.LITTLE ones, since it adds another dimension with the possibility to map threads on different clusters. Another direction is the study of the future ARM SIMD unit called SVE, with up to 2048-bit wide SIMD registers, but which is not yet available. A last direction is to propose a framework to help the developer find the best configuration depending on the performance or energy expectations.

References

[1] P. Estérie, J. Falcou, M. Gaunard, and J.-T. Lapresté, "Boost.simd: Generic programming for portable simdization," in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '14. New York, NY, USA: ACM, 2014, pp. 1–8. [Online]. Available: http://doi.acm.org/10.1145/2568058.2568063

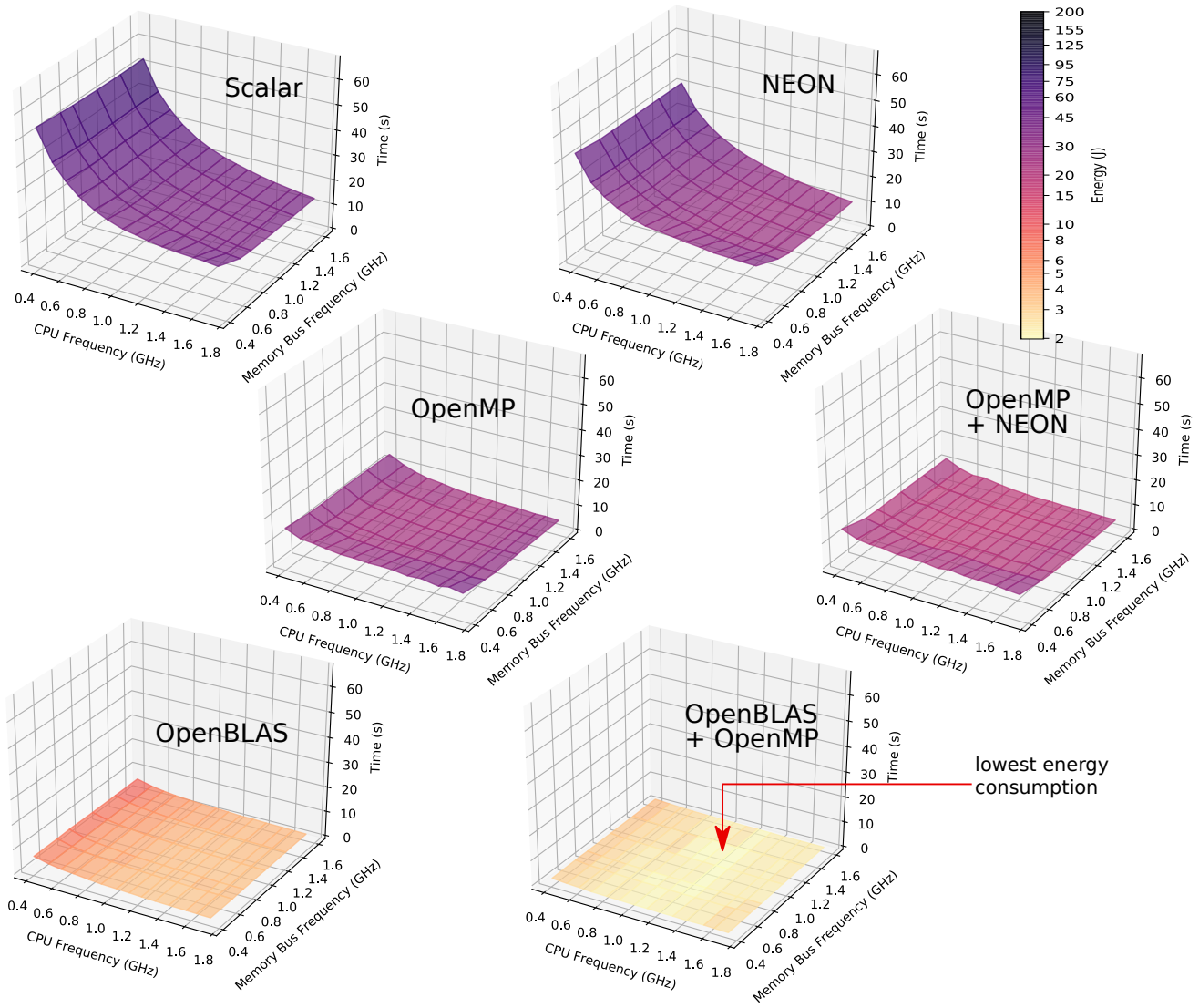

Fig. 9: Performance of the matrix multiplication implementations depending on the CPU and memory bus frequencies. The color represents the energy.

[2] A. Cassagne, O. Aumage, D. Barthou, C. Leroux, and C. Jégo, "Mipp: A portable c++ simd wrapper and its use for error correction coding in 5g standard," in *Proceedings of the 2018 4th Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP'18. New York, NY, USA: ACM, 2018, pp. 2:1–2:8. [Online]. Available: http://doi.acm.org/10.1145/3178433.3178435

[3] T. De Matteis and G. Mencagli, "Proactive elasticity and energy awareness in data stream processing," *J. Syst. Softw.*, vol. 127, no. C, pp. 302–319, May 2017. [Online]. Available: https://doi.org/10.1016/j.jss.2016.08.037

[4] E. Calore, S. F. Schifano, and R. Tripiccione, *Energy-Performance Tradeoffs for HPC Applications on Low Power Processors*. Cham: Springer International Publishing, 2015, pp. 737–748.

[5] E. Calore, A. Gabbana, S. F. Schifano, and R. Tripiccione, "Evaluation of DVFS techniques on modern HPC processors and accelerators for energy-aware applications," *CoRR*, vol. abs/1703.02788, 2017. [Online]. Available: http://arxiv.org/abs/1703.02788

[6] M. Lorenz, P. Marwedel, T. Dräger, G. Fettweis, R. Leupers, and T. U. Dresden, "Compiler based exploration of dsp energy savings by simd operations," in *in Proceedings of the ASP-DAC*. IEEE Press, 2004, pp. 838–841.

[7] J. Coplin and M. Burtscher, "Effects of source-code optimizations on gpu performance and energy consumption," in *Proceedings of the 8th Workshop on General Purpose Processing Using GPUs*, ser. GPGPU-8. New York, NY, USA: ACM, 2015, pp. 48–58.

[8] K. Kasichayanula, D. Terpstra, P. Luszczek, S. Tomov, S. Moore, and G. D. Peterson, "Power aware computing on gpus," in *Proceedings of the 2012 Symposium on Application Accelerators in High Performance Computing*, ser. SAAHPC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 64–73.

[9] M. Danelutto, M. Torquati, and P. Kilpatrick, "A green perspective on structured parallel programming," in *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*, 2015, pp. 430–437. [Online]. Available: https://doi.org/10.1109/PDP.2015.116

[10] M. Danelutto, D. D. Sensi, and M. Torquati, "A power-aware, self-adaptive macro data flow framework," *Parallel Processing Letters*, vol. 27, no. 1, pp. 1–20, 2017. [Online]. Available: https://doi.org/10.1142/S0129626417400047

[11] S. Jubertie, I. Masliah, and J. Falcou, "Data layout and simd abstraction layers: decoupling interfaces from implementations," in *Proceedings of the 2018 International Conference on High Performance Computing & Simulation, HPCS 2018*. IEEE, 2018.