ASSESSING THE PERFORMANCE AND ENERGY USAGE OF MULTI-CPUS, MULTI-CORE AND MANY-CORE SYSTEMS: THE MMP IMAGE ENCODER CASE STUDY

Pedro M.M. Pereira[1], Patricio Domingues[1,2], Nuno M. M. Rodrigues[1,2], Gabriel Falcao[1,3], Sergio M. M. Faria[1,2]

[1]Instituto de Telecomunicações, Portugal
[2] School of Technology and Management, Polytechnic Institute of Leiria, Portugal
[3] Dep. of Electrical and Computer Engineering, University of Coimbra, Portugal

ABSTRACT

This paper studies the performance and energy consumption of several multi-core, multi-CPUs and many-core hardware platforms and software stacks for parallel programming. It uses the Multimedia Multiscale Parser (MMP), a computationally demanding image encoder application, which was ported to several hardware and software parallel environments as a benchmark. Hardware-wise, the study assesses NVIDIA's Jetson TK1 development board, the Raspberry Pi 2, and a dual Intel Xeon E5-2620/v2 server, as well as NVIDIA's discrete GPUs GTX 680, Titan Black Edition and GTX 750 Ti. The assessed parallel programming paradigms are OpenMP, Pthreads and CUDA, and a single-thread sequential version, all running in a Linux environment. While the CUDA-based implementation delivered the fastest execution, the Jetson TK1 proved to be the most energy efficient platform, regardless of the used parallel software stack. Although it has the lowest power demand, the Raspberry Pi 2 energy efficiency is hindered by its lengthy execution times, effectively consuming more energy than the Jetson TK1. Surprisingly, OpenMP delivered twice the performance of the Pthreads-based implementation, proving the maturity of the tools and libraries supporting OpenMP.

KEYWORDS

CUDA, OpenMP, Pthreads, multi-core, many-core, high performance computing, energy consumption

1. INTRODUCTION

Multi- and many-core systems have changed high performance computing in the last decade. Indeed, multi-core CPU systems have brought parallel computing capabilities to every desktop, requiring developers to adapt their applications to multi-core CPUs whenever high performance is an issue. In fact, multi-core CPUs have become ubiquitous, existing not only on traditional laptop, desktop and server computers, but also on smartphones, tablets and in embedded devices.With the advent of GPUs and software stacks for parallel programming such as CUDA[1] and OpenCL[2], a new trend has started, making thousands of cores available to developers[3]. To properly take advantage of many-core systems, applications need to exhibit a certain level of parallelism, often requiring changes to their inner organization and algorithms[4]. Nonetheless, a low to middle range mainstream GPU like the NVIDIA TI 750 delivers a top 1.4 TLFOPs single-precision FP computing power for a price tag below 200 US dollars.More recently, so called System-on-a-Chip (SoC) like the NVIDIA Jetson TK1 and the Raspberry Pi have emerged. Both are quite dissimilar, with Raspberry Pi targeting pedagogical and low cost markets, and Jetson

Tk1 delivering high performance computingto embed systems at affordable prices. More importantly, both systems provide for energy efficient computing, an important topic since the dominant cost of ownership for computing is energy, not only the energy directly consumed by the devices, but also the one used for refrigeration purposes. The quest for performance and computing efficiency is not the sole competence of hardware. In particular, the recent wide adoption of many- and multi-core platforms by developers has been facilitated by the consolidation of software platforms. These platforms have taken away some of the burden of parallel programming, helping developers to be more productive and efficient. Examples include Pthreads[5] for multi-core CPU and OpenMP[6]for multi-core CPU/many-core GPU (OpenMP version 4 or higher is required for targeting GPU), CUDA[1] and OpenACC[7] for NVIDIA GPU devices and OpenCL for CPU, GPU and other accelerators[2].

In this paper, we resort to a compute-intensive image coder/decoder software named Multimedia Multiscale Parser (MMP) to evaluate the performance of several software platforms over distinct hardware devices. Specifically, we assess the sequential, Pthreads and OpenMP versions of MMP over the CPU-based hardware platforms and CUDA over the GPU-based hardware. The assessment comprises computing performance and energy consumption over several heterogeneous hardware platforms. MMP is a representative signal processing algorithm for images. It uses a pattern-matching-based compression algorithm, performs regular and irregular accesses to memory and dictionary searches, uses loops, conditional statements and allocates large amount of buffers. For all these reasons, MMP addresses the major aspects that developers face when programing applications for these architectures. These challenges are common to other signal processing applications that can therefore benefit from the considerations of our study.The CPU-based hardware includes a server with two Intel Xeon E5-2620/v2 CPUs, an NVIDIA Jetson TK1 development board[8] and a Raspberry Pi 2[9]. Regarding GPUs, the study comprises the following NVIDIA's devices: one GTX 680, one Titan Z Black Edition, one GTX 750 Ti and again the Jetson TK1 since it has a 192-core CUDA GPU. The GTX 680, the Titan Z and the Jetson TK1 are based on the Kepler GPU architecture, while the Ti 750 is based on the Maxwell architecture.

Through the assessment of the throughput performance and energy consumption of several multi- and many-core able hardware and software environments, this study contributes for a better knowledge of the behaviour of some platforms for parallel computing. Indeed, a relevant contribution of this work is the assessment of two embedded platforms: the Jetson Tk1 development board and the Raspberry Pi 2. This study confirms that the Jetson TK1 development board with its quad-core CPU and CUDA-able GPU is an effective platform for delivering high performance with low energy consumption. Conversely, the Raspberry Pi 2 is clearly not appropriate for high performance-bounded operations. Another contribution of this work lies in the comparison between the use of the paradigms OpenMP and Pthreads to solve the same problem, with a clear performance advantage for OpenMP. This study also confirms the need for different parallelization approaches, depending whether multi-core/multi-CPUs or many-core systems are targeted. Finally, it also shows that speedups, albeit moderate, are possible to attain even with applications that are challenging to parallelize.

The paper is organized as follows. Section 2 reviews related work. Section 3 presents the hardware and parallel paradigms targeted in this work. Section 4 outlines the MMP algorithm, while Section 5 presents the main results. Finally, Section 6 concludes the paper and presents future work.

2. RELATED WORK

Since the Jetson development boardsare relatively recent, scientific studies regarding their performance and energy consumption are still scarce. Paolucci et al. analyse performance vs. energy consumption for a distributed simulation of spiking neural networks[10]. The comparison involves two Jetson TK1 development boards connected through ethernet and a multiprocessor system with Intel Xeon E5-2620/2.10 GHz, while the application is based on the Message Passing Interface standard (MPI)[11]. The authors report that performance-wise, the server system is 3.3 times faster than the parallel embedded system, but its total energy consumption is 4.4 times higher than the dual TK1 system. In[12], the authors evaluate the RX algorithm for anomaly detection on images for several low-powered architectures. The study assesses systems based on general processors from INTEL (Atom S1260 with two cores) and ARM (Cortex-A7, Cortex-A9, Cortex-A15, all quad-core systems); and two low-power CUDA-compatible GPUs (the 96-core Quadro 1000M and the 192-core GK20a of Jetson TK1). As a reference, they use an Intel Xeon i7-3930 CPU with no accelerators. They report that for the IEEE 754 real double-precision arithmetic RX detector, the Jetson TK1 system yields an execution time close to the reference desktop system, using one tenth of the energy.Fatica and Phillips report on the port and optimization of a synthetic aperture radar (SAR) imaging application on the Jetson TK1 development board[13]. The port involves the adaptation of the Octave-based applicationto CUDA. Through several software optimizations, the execution time of the application is brought down from 18 minutes to 1.5 seconds, although the main performance improvements come from refactoring the code, and not from using the Jetson TK1 GPU through CUDA.

The Glasgow Raspberry Pi cloud project reports that the 56-Raspberry Pi data center solely consumes 196 Wh (3.5 Wh per system), while a real testbed would require 10,080 Wh (180 Wh per system), that is, roughly, 50 times more[14].Similarly, Baunthoroughly studies the performance of several clusters comprised of SoC boards: RPi-B, RPi2-B and the Banana Pi[15]. The author concludes that the studied cluster of RPi2-B provides 284.04 MFLOPS per watt, which would be sufficient for 6th place in the November 2015 Green 500 list if solely the performance per watt is considered. Additionally, these low cost and low maintenance clusters are interesting for several academic purposes and research projects.

Since the inception in the 2000s of multi-core and many-core systems, a significant volume of scientific literature has been produced, often comparing the performance of both types of systems. Lee et al.[16] report that a regular GPU is, on average, 14x faster than a state-of-the-art 6-core CPU over a set of several CPU- and GPU-optimized kernels. Bordawekar et al. study the performance of an application that computes the spatial correlation for large images dataset derived from natural scenes[17]. They report that the optimized CPU version of the application requires 1.02 seconds on an IBM power 7-based system, 1.82 seconds on an Intel Xeon, while the CUDA-based version runs on 1.75 seconds over an NVIDIA GTX285. Stamatakis and Ott report on a performance study on the bioinformatics field involving OpenMP, Pthreads and MPI[18]. They use the RAxML application that studies large-scale phylogenetic inference. The authors mention some numerical issues with reduction operations under OpenMP due to the non-determinism of the order of additions. We encountered a similar situation in our initial adaptation of the code, where the determinism of the sequential version could not be reproduced on the parallel version, yielding slightly different final results. Regarding performance, the authors report better scalability of OpenMP relatively to Pthreads on a two-way 4-core Opteron system (8 cores) using the Intel C Compiler (ICC) suite.

3. COMPUTING ENVIRONMENTS

Next, we describe the hardware and software environments used in this study.

3.1. HARDWARE

We present the hardware used in the experiments, namely, the Xeon-based server and the GPUs, and the energy consumption measurement hardware.

3.1.1. SERVER SYSTEM

All the tests requiring a server system were performed on a machine with a two-way Intel Xeon E5-2620/v2 CPUs, clocked at 2.10 GHz. Each physical core has a 32 KiB L1 cache for data, 32 KiB L1 cache for instructions, plus a unified 256 KiB level 2 cache. Additionally, all the physical cores share a 15MiB on-chip Level 3 cache memory. Each CPU holds 6 physical cores that are doubled through SMT hyper-threading. Therefore, in total, the desktop testing machine has 12 physical cores (6 core per CPU) that yield 24 virtual cores.

3.1.2. DISCRETE GPUS

The CUDA-based tests involving discrete GPUs were conducted with a GTX 680, a Titan Black Edition and a GTX 750 Ti, all from NVIDIA. Both the GTX 680 and the Titan Black Edition are Kepler-based GPU, while the GTX 750 Ti is based on the Maxwell architecture. All of them were used through the PCI Express interface in the Xeon E5-2620/v2 server. The main characteristics of the GPUs are summarized inTable 1.

Table 1. Main characteristics of the GPUs (TFLOPS are for 32bit FP)

	GTX 680	Titan Black	GTX 750 Ti	Jetson TK1
CUDA cores	1536	2880	640	192
Mem. (DDR5)	2 GiB	6 GiB	2 GiB	2 GiB
Mem. width(bits)	256	384	128	64
Power (watts)	195	250	60	14
TFLOPS	3.090	5.121	1.306	0.300
Architecture	Kepler	Kepler	Maxwell	Kepler

3.1.3. THE JETSON TK1 DEVELOPMENT BOARD

The Jetson TK1 development board is a SoB (System on a Board) implementation of the NVIDIA's Tegra TK1 platform. It combines a 32-bit quad-core ARM cortex A15 CPU with a Kepler-based CUDA-able GPU[19]. The CPU is classified by NVIDIA as a 4-PLUS-1 to reflect the ability of the system to enable/disable cores as needed for the interest of power conservation[20]. For this purpose, the CPU has 4 working cores and a low performance/low power usage core. The low performance core, identified as the ***PLUS 1***, drives the system when the computational demand is low. When the computing load increases, the other cores are activated as needed. The system also balances the computing power versus the power consumption by varying the memory operating frequency and disabling/enabling support for I/O devices like USB and/or HDMI ports. The Jetson TK1 development board has a single CUDA multiprocessor (SMX) with 192 CUDA cores, with 64 KB shared memory and a 64-bit wide memory bus. The device has 2 GiB of RAM, which are physically shared between the CPU and the GPU. The main details for the GPU of the Jetson TK1 are listed in Table 1.

Although the Jetson TK1 development board allows for a large range of performance modes due to the possibility of controlling the GPU frequency – it can be varied by steps from 72 MHz to 852 MHz – weonly consider two performance modes: i) *low power* and *ii*) *high performance*. The *low power* mode puts the system in low power at the cost of performance, settingthe GPU frequency to its minimum of 72MHz. Conversely, in *high performance* mode, the GPU is sets at 852 MHz, while all other systems are also set for top performance. Interestingly, even when set

for maximum performance, only the needed hardware modules of the Jetson TK1 are enabled. For instance, when running a CPU-bound application that does not use the GPU, the system does not enable the GPU.

3.1.4. RASPBERRY PI 2

The Raspberry Pi is a low cost, low power single board credit-sized computer developed by Raspberry Pi Foundation[9]. The Raspberry Pi has attracted a lot of attention, with both models of the first version – model A and model B – reaching sales in the order of millions. Major contributor for its popularity has been the low prices and the ability to run the Linux OS and its software stack. The version 2 of the Raspberry Pi, which is the one used in this study, wasreleased in 2015. Model B – the high end model of the Raspberry Pi 2 – has a quad-core 32-bit ARM-Cortex A7 CPU operating at 900 MHz, a Broadcom VideoCore IV GPU and 1 GiB of RAM memory shared between the CPU and the GPU. Besides the doubling of the RAM memory, an important upgrade from the original Raspberry version lies in the CPU which has four cores and thus can be used for effective multithreading. Each CPU core has a 32 KiB instruction cache and a 32 KiB data cache, while a 512 KiB L2 cache is shared with all cores. The CPU implements the version 7 of the ARM architecture, which means that Linux distributions available for the ARM v7 can be run on the Raspberry Pi 2. The GPU is praised for its capability in decoding video with resolution of up to 1080 pixels (full HD) supporting the H.264 standard[21]. However, to the best of our knowledge, no standard parallel programming interfaces like OpenMP 4 and OpenCL are available for the GPU of the Raspberry Pi. Although the Raspberry provides for six different performance modes, we solely consider two of these modes. The *low power* mode corresponds to the *None* mode of the Raspberry Pi 2, with the ARM CPU sets to 700 MHz, the cores to 250 MHz and the SDRAM to 400MHz. The *high performance* mode increases the ARM CPU to 1000 MHz, the cores to 500 MHz and the SDRAM to 600 MHz. It corresponds to the *Turbo* mode of the Raspberry Pi 2. The main characteristics of both the Jetson TK1 and the Raspberry Pi 2 are shown inTable 2. Table 3 displays the memory bandwidth measured on copies between non-pageable RAM (host) and the GPUs (devices) and vice-versa. The values were measured with the *bandwidthTest* (NVIDIA SDK).

Table 2. Main characteristics of the Embedded Systems.

Device	CPU cores	GPU cores	TFLOPS (32-bit FP)
Jetson TK1	4+1 ARM-v7	192 (CUDA)	0.300
Raspberry Pi 2	4 ARM-v7	n.a.	0.244

Table 3. Measured memory bandwidth.

Device	Host to Device (MB/s)	Device to Host (MB/s)
GTX 680	6004	6530
Titan Black	6119	6529
Jetson TK1 (LP)	997	997
Jetosn TK1 (HP)	6380	6387

3.2. SOFTWARE

We briefly present the software frameworks OpenMP, Pthreads and CUDA.

3.2.1. OPENMP

OpenMP (Open Multi Programming) is a parallel programming standard for shared memory computers available for the C, C++ and Fortran programming languages. Although the standard appeared in 1997, the emergence of multi-core CPUs have contributed to renewed interest in

OpenMP. The standard is driven by the OpenMP ARB consortium[22]. The main goal of the standard is to provide a set of high level constructs that allows programmers to identify zones oftheir source code that can be parallelized. For instance, the programmer marks a given section of the code (e.g., a loop) for parallelization, distinguishing, among many other things, between private and shared variables and how the section should be split. The distinction between shared/private variables allows OpenMP to properly deal with concurrency issues, while the split indication provides OpenMP guidance on how the underlying working threads should be organized. The high level constructs of OpenMP comprise compiler directives, functions and environment variables. Through all these input options, programmers can pinpoint to the compiler the parallel zones of their code.

3.2.2. POSIX THREADS

POSIX Threads (henceforth Pthreads) is a POSIX standard for threads[23]. It defines an API for threads providing the definition of data structures and functions for the manipulation and synchronization of threads, while the implementation details are left to the discretion of implementers. Pthreads is widely supported, and it is often the first choice for dealing with threads in UNIX platforms.Contrary to the high level paradigm made available by OpenMP, Pthreads is a low level approach, requiring the programmer to explicitly handle the creation, synchronization and destruction of threads.

3.2.3. CUDA

NVIDIA's Compute and Unified Device Architecture (CUDA) is a proprietary parallel programming platform that targets exclusively GPUs from NVIDIA[3]. It first appeared in 2007 and has since evolved, with a new release occurring approximately every year. The main goal of CUDA is to facilitate the use of GPUs for many-core programming in an efficient manner. For this purpose, CUDA provides a set of abstractions such as threads, a 3D set of coordinates and kernels, as well as an API that provides, among other things, data transfers between the machine that hosts the GPU(s) and the GPU itself[24].A CUDA *kernel* is an entry point to GPUs and appears as a function to the CUDA programmer. Within the kernel, the programmer specifies the operations to be performed by CUDA threads that run on the GPU. Note that CUDA threads are substantially lighter and different than common OS threads, like the ones available through Pthreads-based systems. In fact, a GPU can easily support thousands of threads, with threads implicitly created whenever a kernel is launched and internally mapped to the CUDA cores of the executing GPU. From the programmer point of view, a kernel launch involves the specification of the execution geometry, which comprises two main entities: grid and blocks. A block contains up to three dimensions of threads, while a grid holds the blocks of threads, again in a 3D organization. For example, if called with a (3,2) blocks within a (4,5) grid, a kernel will be executed with 6 blocks laid out on a 3x2 2D grid, with each of the six blocks having 20 threads distributed over a 4x5 2D geometry, totalling 120 threads as shown in Figure 1.

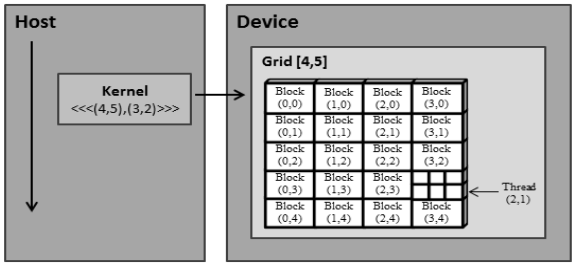


Figure 1. Execution grid with 4x5 2D blocks, each block having 3x2 threads.

Within the GPU code, CUDA provides a set of identifiers that allows for the localization of the current thread within a block (threadID.x, .y and .z) and of the current block (blockID.x, .y and .z). Through these identifiers, the programmer can assign a particular zone of the dataset to each thread. For example, the addition of two matrices can be performed by creating a 2D execution geometry with the dimension of the matrices, whereas each thread performs the addition of the corresponding pair of parcels of the matrices. This way, the addition is performed in parallel. For matrices larger than the maximum dimensions of the execution geometry, each thread can be looped around, performing an addition and then moving to the next assigned pair of parcels. Regarding memory, CUDA distinguishes between the *host memory* and the *device memory*. The former is the *system RAM*, while the latter corresponds to the memory linked to the GPU. By default, CUDA code running within a GPU can only access the GPU memory. Proper memory management is important in CUDA and can have deep impact in performance[25][24].CUDA's software stack includes compilers, profilers, libraries and a vast set of examples and samples. From the programming language point of view, CUDA extends C++ and C through the addition of a few modifiers, identifiers and functions. Nonetheless, the logic and semantic of the original programming language is preserved. In this study, CUDA was used in a C environment.

4. THE MULTIMEDIA MULTISCALE PARSER

The Multidimensional Multiscale Parser (MMP) is a pattern-matching compression algorithm that has been mainly developed for image and video coding[26]. Although lossless and lossy versions of the algorithm exist, for this paper we solely used the lossy version to encode/decode images.The MMP algorithm relies on pattern matching to replace input blocks by a codevector, which belongs to an adaptive dictionary. The compression efficiency results from replacing a large quantity of pixels by one single index representing the chosen codevector. For image coding, MMP divides the input image into square blocks with 16×16 pixels, which are processed sequentially. For each 16×16 block, an exhaustive searching procedure is used to find the codevector that best matches the input block. After determining the approximation for the original scale, MMP segments the block into two parts and repeats the searching procedure using a scaled version of the codebook, checking the matching accuracy when the block is divided into segments of different scales, that is, blocks of different dimensions. Indeed, for a given input block, all combinations of subblocks whose dimensions are a power of 2 are analysed. For instance, a 16×16 pixels block can be split in two 8×16, two 16×8, four 4×4 blocks and so on, up to the smallest possible block, which is a single pixel (1×1).

The criteria used by the MMP algorithm to select the best approximation is the so-called Lagrangiancost $\boldsymbol{J}$, given by the equation $J = D + \lambda \cdot R$, where $\boldsymbol{D}$ measures the distortion between the original block and the tested codevector, and $\boldsymbol{R}$ represents the number of bits (rate) needed to encode the codebook element. The value of $\boldsymbol{\lambda}$ is an input parameter which is set before the encoding starts and which remains constant throughout the coding phase. It tunes the compromise between bitrate usage and the quality of the compressed image. Higher values of λcause a penalty in the rate value, favouring high compression ratios (e.g., less bits are needed to encode a block), and causing higher distortion values or lower quality images. Conversely, a low λ benefits quality, by increasing the importance of the distortion term for the computation of J, at the cost of a higher bitrate requirement. For lower values of λ, one observes an increase in the size of the codebook with a strong impact on computational complexity.The distortion between two equally-sized blocks is computed as Mean Squared Error (MSE) of the difference between the intensity value of the pixels of one block and the corresponding pixels of the candidate block. Specifically, the distortion $\boldsymbol{D}$ between an input block $\boldsymbol{B}$ and a given element $\boldsymbol{C}$ of the codebook is defined by Equation 1, where $M \times N$ represents the size of boththe$\boldsymbol{B}$ and$\boldsymbol{C}$blocks.

$$D = \sum_{i=1}^{M}\sum_{j=1}^{N}\left(B(i,j) - C(i,j)\right)^2$$

Equation 1. The distortion between two equally-sized blocks.

For a given block, the best fit corresponds to the block or set of subblocks that yield the lowest J. The distortion is handled through single floating point IEEE 754 format. The recurrent optimization of an input block works by segmenting it into two sub-blocks, each with half the pixels. The two halves are then recursively optimized using new searches in the dictionary. The decision to segment a block is made by comparing the sum of the costs for each half with the cost for approximating the original block.The need to compute the distortion $\boldsymbol{D}$ among a vast set of blocks and the input block is the main cause for the high computational load of the algorithm. For example, the single-threaded MMP requires around 2000 seconds to encode the 512×512 pixels 8-bit gray level Lenna image when run on an Intel Xeon E5-2620/v2 machine.

Every time MMP segments a block, a new pattern is created by the concatenation of two smaller codewords. This new pattern is then inserted in the dictionary, allowing for future uses in the coding procedure. Furthermore, scale transformations are used in order to adjust the dimensions of the vector and create new patterns that can be used to approximate future blocks to be coded with any possible dimensions[27][26].

Another relevant feature of MMP is the use of a hierarchical prediction scheme, similar to the one used by H.264/AVC video encoding standard[28]. For each original block,B, a prediction block, P, is determined using the previously encoded neighbouring samples, located to the left and/or above the block to be predicted. A residue block can then be computed by using a pixel-wise difference: $Q = B - P$. This allows the use (encoding) of the residue block Q instead of B, since the decoder is able to determine P and compute $B' = P + Q'$, where B' and Q' represent the encoded (approximated) versions of B and Q, respectively.By using different prediction models, the residual patterns Q tend to be more homogeneous than the original image patterns. These homogeneous patterns are easier to learn, thus increasing the efficiency of the dictionary and of the approximation of the encoded blocks, resulting in a more efficient method. Figure 2 presents three examples of available prediction modes (vertical, horizontal and diagonal down/right) and, at the bottom right, all possible prediction directions. These prediction modes are available for both MMP and H.264/AVC[28].

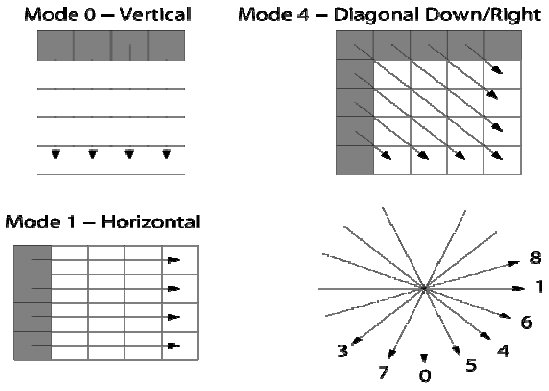


Figure 2. Prediction modes (0, 1, 4) and all possible prediction directions.

MMP uses a hierarchical prediction scheme, meaning that block of different dimensions can be used in the prediction process (16 x 16 down to 4 x 4). For each possible prediction scale, MMP

tests all available prediction modes and selects the one with the best result. This full search scheme enables MMP to choose not only the most favourable prediction mode, but also the best block size to be used in the prediction step. As a result, MMP becomes highly flexible and has a relevant performance improvement, but at the cost of an exponential complexity increase, related with the many new coding options which have to be tested.

4.1. PARALLELIZATION STRATEGY

We now describe the parallelization strategies for MMP. We first focus on the strategy employed for OpenMP and Pthreads, and then on the approach used for the CUDA version.

To further increase compression ratio, MMP resorts to a prediction module. This module aims to predict the neighbour pixels of a given block and bears close resemblance to the intra-prediction schemes of others compression algorithms like H.264/AVC[28] and H.265/HEVC[29]. Prediction in MMP resorts to previously coded blocks of the neighbourhood of the current block. Specifically, MMP uses up to 10 different neighbourhood patterns for predictions. Since these predictions can be computed simultaneously, they can be parallelized. In fact, this is the followed approach for the CPU-based parallel version of MMP, where the prediction module is multi-threaded either directly through Pthreads or indirectly through OpenMP directives. This ability for CPU-level parallelization is one of the reason for selecting MMP as a benchmark for assessing performance and energy consumption across several hardware and software platforms. Note however, that MMP is an inherently sequential algorithm, since the encoding of an input block is dependent on the codebook. Indeed, the codebook is updated at the end of the encoding of each input block with the new blocks that might have arisen during the encoding of the block. This means that the encoding of input block $N + 1$ can only proceed after block N has been processed, thus departing MMP from traditional image algorithms that exploit parallelism by processing multiple input blocks at once. In MMP, the parallelism that can be exploited is limited to the encoding operations that occur within the processing of each single input block. Nonetheless, speedups for MMP can still be achieved through OpenMP and Pthreads software stacks.Regarding the MMP encoding application, OpenMP was used to parallelize the 10 prediction modes, effectively allowing the simultaneous execution of the 10 modes. Although this restricts the parallelism to 10, it was the only effective approach to yield speedup from MMP with OpenMP.

The CUDA version of MMP (henceforth CUDA-MMP) relies on the following three main CUDA kernels: *kernelDistortion*, *kernelReduction* and *kernelSearchCodebook*. The first kernel computes the Lagrangian between the input block and all the candidate blocks. The second kernel determines the candidate block that has the lowest Lagrangian. The *kernelDistortion* and *kernelReduction* act in pairs, with *kernelReduction* being called right after *kernelDistortion* has produced a set of candidate blocks. Finally, *kernelSearchCodebook* intervenes at the end of the processing of an input block. It searches the codebook for an equivalent block to the one (or set of blocks) that was computed for the current input block. An equivalent block is a block whose distortion falls within a given radius to the candidate block. If an equivalent block is found, the candidate block is discarded, with MMP instead selecting the equivalent one. Table 4 shows the number of calls per kernel for the encoding of the 8-bit gray512x512 pixels Lenna image.

Table 4. Number of kernel calls (Lenna image).

kernel	# of calls
kernelDistortion	667805
kernelReduction	667805
kernelSearchCodebook	1024

5. MAIN RESULTS

We now discuss main results. We present the configuration for the experimental tests, and then, analyse the most relevant results regarding execution time and energy consumption.

5.1. CONFIGURATION

Each testwas run 20 times, except for the Raspberry Pi 2, where solely 10 executions were performed per test, due to its slower speed. As the standard deviation values are close to zero, we only report the average of the execution times. The tests consisted in performing the MMP encode operation of the Lenna image in a 512x512 8-bit gray format. The λ quality parameter of MMP was set to 10, a good balance between quality and output bitrate.

5.1.1 OPERATING SYSTEM AND TOOLS

For each platform, the following operating systems and compiler tools were used:

- Xeon E5-2620/v2: Ubuntu 14.10, kernel 3.13.0-39 SMP, gcc 4.8.2, CUDA driver 340.58, nvcc 6.5.12
- Jetson TK1: 32-bit tegra-ubuntu, kernel 3.10.40, gcc 4.8.2, nvcc 6.5.12
- Raspberry Pi 2: 32-bit raspbian, kernel 3.18.11-v7+SMP, gcc 4.6.3

5.1.2 MEASURING POWER DEMAND AND ENERGY CONSUMPTION

The power consumption measurements were performed with an external device that has a board based on the Microchip MCP39F501[30]. The power measurement device is directly connected to the power wall and provides a power socket where the power cord of the measured system is connected. It thus acts as a device in the middle that measures the power consumption of the device being monitored. The measurement device is attached to a controlling computer through its USB port. Measurements are periodically logged with a timestamp that are used, in a post processing stage, to match the power measurements with the activity of the monitored device. For all the experiments of this study, the power consumption periodicity was set to 20 seconds, a balance between the execution length and the number of samples produced by a run of MMP.

5.2. CPU-BASED RESULTS

In this section, we assess the CPU-based solutions. First, we separately analyse the sequential-, OpenMP- and Pthreads-based versions and then compare the OpenMP and Pthreads versions. Finally, we analyse the GPU-based versions.

5.2.1 SEQUENTIAL

The so-called sequential version of MMP is highly optimized, but it solely uses one CPU thread, and therefore it does not benefit from multi-core CPUs nor from GPUs. The execution time results and the power consumption measurements for the sequential version are shown on Table 5. The column *Exec. times* shows the execution times in seconds. The column *Speed (ratio)* corresponds to the ratio between the speed of a device and the speed of the dual Xeon E5-2620/v2, which is used as reference. Note that the speed of a device is computed as the inverse of the time needed for the device to execute MMP. The column *Avg. Power (watt)* displays the average of the power measurements that were performed along the execution. The column *Energy (Wh)* aims to quantify, in watt-hour, the amount of energy consumed by the device. It corresponds to the product of *avg. power usage* times *execution time*divided by the 3600 seconds of an hour.

The *Energy (ratio)* corresponds to the power usage ratio between the current device and the reference system, again the Xeon E5-2620/v2 server. Finally, the *Efficiency (ratio)* corresponds to the ratio between the *speed ratio* and the *power ratio*. It measures the efficiency of the device running MMP against the E5-2620/v2 reference.

The efficiency metric shows that the Jetson TK1 is the most appropriate device when considering execution speed and energy consumption, with an efficiency ratio in the nearby of 4.8, with a slight advantage for the *low power* over the *high performance* mode. Note that the efficiency ratio of the Jetson TK1 is due to its low energy consumption (roughly 7.2 watt-hour) which is less than 1/10 of the energy needed by the Xeon E5-2620/v2, while its execution speed for Sequential-MMP is approximately 0.46 of the speed achieved by Xeon E5-2620/v2 server. At the other extreme of the scale, the *low power* mode of the Raspberry Pi 2 delivers half the efficiency of the reference system. In fact, the raspberry consumes more energy than the Jetson TK1 (13.240 Wh vs. 7.183Wh), since its lower instantaneous power usage is overshadowed by the fact that it takes roughly 5 times longer to execute the MMP encoding operation than the Jetson TK1. Interestingly, at least for the single-threaded CPU version of MMP, there seem to be no meaningful differences on the Jetson TK1 between the low power mode and the maximum performance mode, while only a marginal difference exists on the execution time between the *low power* and the *high performance* mode of the Raspberry Pi 2 (24826.195 vs. 24442.232 seconds). This is mostly due to the memory-bound nature of sequential-MMP, where a faster CPU does not meaningfully improve execution time due to saturation at the CPU/RAM traffic level.

Table 5. Execution times and power usage for the sequential version of MMP.

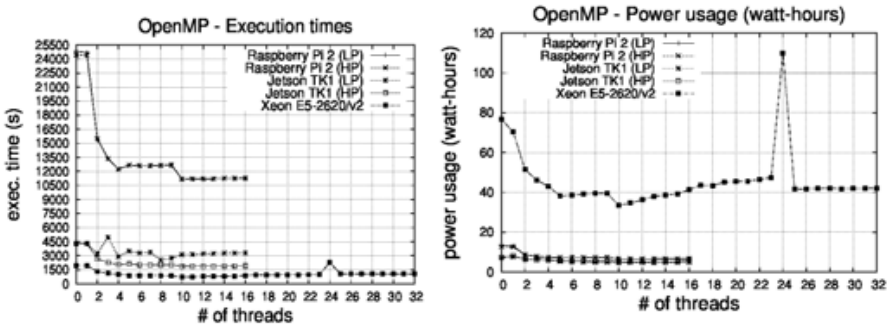
Sequential	Exec. Time (seconds)	Speed (ratio)	Avg. Power (watts)	Energy (Wh)	Energy (ratio)	Efficiency (ratio)
Xeon E5-2620	1967.813	1	140.191	76.630	1	1
Jetson TK1 LP	4297.038	0.458	6.018	7.183	0.094	4.872
Jetson TK1 HP	4290.311	0.459	6.086	7.253	0.095	4.832
RPi 2 LP	24826.195	0.079	1.920	13.240	0.173	0.457
RPi 2 HP	24442.232	0.081	1.879	12.762	0.167	0.485

5.2.2 OPENMP

The OpenMP version of MMP (henceforth OpenMP-MMP) was run with a number of working threads ranging between 1 to 16 on the Jetson TK1 board and Raspberry Pi 2 and 1 to 48 on the Xeon E5-2620/v2. The rationale is that both the Jetson TK1 and Raspberry Pi 2 have a quad-core CPU, while the E5-2620 server system has two E5-2620/v2 hexa-core CPUs, totalling 24 virtual cores. However, since in all experiments no behaviour change was observed past 32 threads, we only present results up to 32 threads for the Xeon E5-2620/v2 server.The execution times for OpenMP-MMP across all studied platforms are plotted inFigure 3a. The X-axis represents the number of OpenMP working threads, that is, the number of threads excluding the main thread (the main thread does not perform any computation). The slot *thread 0* corresponds to the execution time of Sequential-MMP and aims to ease comparisons.

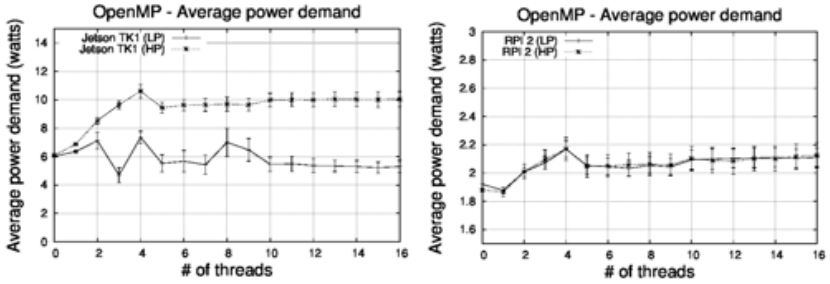
For the E5-2620/v2 server, OpenMP-MMP attains its minimum execution time with 719.287 seconds when the execution is performed with 10 working threads, thus being 2.74 faster than the single-threaded sequential version. The 10-thread performance barrier matches the underlying algorithm used to adapt MMP to OpenMP, where parallelization is performed along the 10 prediction modes as previously seen on Section 4.1. Increasing the number of threads beyond 10 degrades the execution times (895.727 seconds for 16 threads and 1072.816 seconds for 32 threads), mostly due to the overhead of having more threads, while the parallelism has been exhausted. However, an unexpected disruption occurs with 24 working threads, with the

execution times worsening to 2283.686 seconds, which is the double of both the 23-thread execution (994.851 seconds) and 25-thread execution (1065.163 seconds). Note that the peak on the execution times coincides with the saturation of the 24-virtual core E5-2620/v2 server, since 24 working threads corresponds to 25 threads plus the OS regular activity. After checking the source code of the used OpenMP implementation, we confirmed that it enforces CPU affinity, assigning one thread per core, whenever the number of threads is less or equal the number of (virtual) cores of the underlying system. Thus, when the number of threads is equal to the number of cores, one of the cores used by OpenMP is also necessarily used by the operating system, thus disturbing the balance of the OpenMP execution. Moreover, due to the natural



(a) Execution times (b) Energy consumption

Figure 3. Main results for OpenMP-MMP.



(a) Jetson TK1 (b) Raspberry Pi 2 (LP/HP)

Figure 4. Average power demands for OpenMP-MMP (JTK1 and RP2).

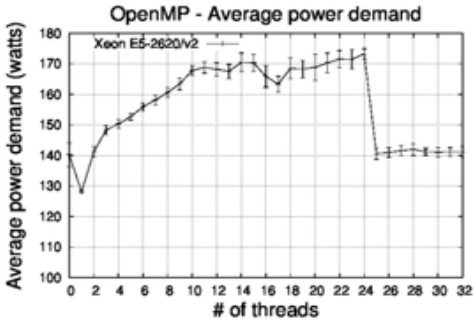


Figure 5. Average power demands for OpenMP-MMP (Xeon E5-2620/v2).

OpenMP organization of the execution in fork/join sections, a delayed thread impacts a whole section, with each delayed section accumulating on the execution time. This hypothesis is further confirmed by the fact that with 25 or more threads, the execution time of OpenMP-MMP regains its slow degradation observed before the 24-thread peak, due to the fact that OpenMP no longer enforces CPU affinity. This is further confirmed by the average power demands (Figure 5), which drops from 173.1981 to 140.6071 watts, indicating that the number of active cores drops sharply after the 24-thread peak.

For the Raspberry Pi 2, the fastest execution time is 11177.974 seconds, achieved with 10 working threads, meaning that the OpenMP version of MMP is roughly 2.19 times faster than the sequential-MMP version ran on the Raspberry Pi 2. Beyond 10 threads, the execution times slowly degrades reaching 11280.480 seconds with 16 working threads. As can be seen on the plot, where the curves for *low power* and *high performance* modes match, the execution times difference between the two modes are marginal.The behaviour for the Jetson TK1 is strongly influenced by the running mode. Indeed, in performance mode, the Jetson TK1 achieves its fastest execution time of 1888.244 seconds for OpenMP-MMP, when ran with 10 working threads. Beyond this threshold, the execution time slowly degrades (1915.380 seconds for 16 working threads). This matches the behaviour of the other platforms. However, the low power mode has a substantially different behaviour: it achieves its fastest execution of 2539.583 seconds with 8 working threads, which corresponds to two threads per physical core. Moreover, contrary to the other platforms and the *high performance* mode, the evolution of the execution times in scenarios with less than 8 threads is not linear. For instance, while two threads yields 3234.894 seconds, the execution with three threads requires 4979.784 seconds, and the execution with four threads -- matching the number of cores -- only takes 2906.157 seconds. This pinpoints that in *low power* mode, the performance achieved with OpenMP on the Jetson TK1 are strongly dependent on the number of threads. In particular, the best results seem to be achieved with four and eight threads.

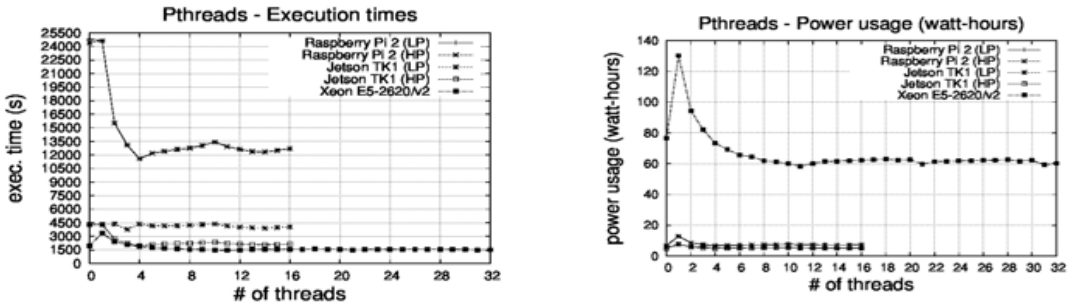
The energy consumption results are shown onFigure 3b. The Raspberry Pi 2 yields a similar behaviour for both the low power and high performance mode, consuming around 6.5 watt-hours for the OpenMP-MMP version. Relatively to the power demand, the Raspberry Pi 2 has a stable behaviour for both execution modes, increasing slightly with the number of threads. Moreover, there is practically no differences between the two execution modes. Indeed, for the execution of OpenMP-MMP, the average power demand ranges from 1.877 watts (one worker thread) to 2.110 watts (16 threads) for the *low power* mode, and from 1.866 watts (one thread) to 2.125 watts for the *high performance* mode. The energy consumption of the Raspberry Pi 2 is shown on Figure 3b along with the energy consumption of all the other devices. The average power demand is plotted in Figure 4. The Xeon E5-2620/v2 server attains its best performance with 10 working threads consuming 33.528 watt-hours, while the highest consumption occurs at the aforementioned ill 24-thread execution with 109.870 watt-hours. Note that the one-thread OpenMP-MMP consumes less energy than the sequential execution (128.114 vs. 140.191 watt-hours), requiring less average power (Figure 5). This is most probably due to the strict core-affinity policy enforced by OpenMP that aims to keep each thread in the same core along the whole run. This minimizes the number of used cores, allowing for non-used cores to remain idle in low power mode.The high performance mode of the Jetson TK1 only consumes slightly more energy than the low power mode: 5.236 vs. 4.936 watt-hours, while the execution times are significantly different (1888.244 vs. 2539.583 seconds). The plot for the average power usage of the Jetson TK1 (Figure 4a) shows that the power demand varies widely for the low power mode, possibly due to individual CPU cores being activated/deactivated in response to the system load.

Table 6 summarizes the best results for OpenMP-MMP. While the Xeon E5-2620/v2 server provides for the fastest execution, the Jetson TK1 yields the best efficiency ratio with 15.324 in *high performance* mode and the lowest overall energy consumption with 4.936 watt-hours for the

low power mode. While it only requires an average power of 2 watts, overall the Raspberry Pi 2 consumes more energy than the Jetson TK1, since it takes much longer to perform the MMP encoding operation.

Table 6. Execution times and energy consumption for OpenMP-MMP.

Pthreads	#threads	Exec. Time (seconds)	Speed (ratio)	Avg. Power (watts)	Energy (Wh)	Energy (ratio)	Efficiency (ratio)
Xeon E5-2620 (sequential)	-	1967.813	1	140.191	76.630	1	1
Xeon E5-2620 (OpenMP)	10	719.287	2.736	167.806	33.528	0.438	6.247
Jet.TK1 (LP)	8	2539.583	0.775	6.997	4.936	0.064	12.109
Jet. TK1 (HP)	10	1888.244	1.042	9.983	5.236	0.068	15.324
RPi 2 LP	10	11185.531	0.176	2.093	6.503	0.085	2.071
RPi 2 HP	10	11177.974	0.176	2.103	.530	0.085	2.071



(a) Execution times (b) Energy consumption

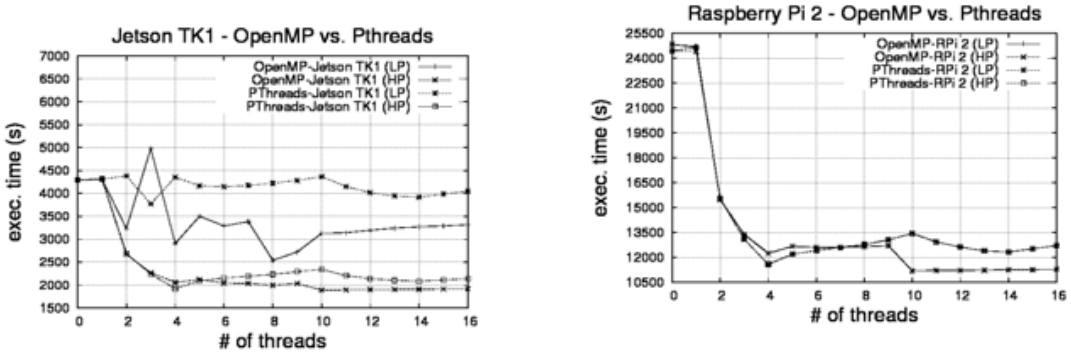
Figure 6. Main results for Pthreads-MMP.

5.2.3 PTHREADS

The execution times for Pthreads-MMP are shown in Figure 6. Once again, practically no differences exist between the two studied modes of the Raspberry Pi 2. Both modes yield their fastest execution times close to 11580 seconds with 4 working threads. Relatively to the sequential execution, this corresponds to a 2.1 speedup. The fastest average execution time for the Pthreads-MMP on the Xeon E5-2620/v2 server is 1458.617 seconds, achieved with 11 working threads. This corresponds to a 1.35 speedup relatively to the sequential version.Surprisingly, Pthreads-MMP executes practically within the same execution times of Sequential-MMP when the Jetson TK1 is set for *low power*. The only exception occurs with three working threads, when it achieves a marginal speedup of 1.13 relatively to the sequential single-thread version. However, when set to *performance mode*, the Jetson TK1 achieves a speedup of 2.23 with 4 working threads. In fact, with 1924.222 seconds, the Jetson TK1 *high performance* is only 25% slower than the Xeon E5-2620/v2 for Pthreads-MMP (1924.222 vs. 1458.617 seconds) and slightly faster than the execution of Sequential-MMP on the Xeon server (1924.222 vs 1967.813 seconds), all this with a fraction of the energy consumption. The main performance and energy consumption results of Pthreads-MMP are grouped in Table 7.

Table 7. Execution times and energy consumption for Pthreads-MMP.

Pthreads	#threads	Exec. Time (seconds)	Speed (ratio)	Avg. Power (watts)	Energy (Wh)	Energy (ratio)	Efficiency (ratio)
Xeon E5-2620 (sequential)	-	1967.813	1	140.191	76.630	1	1
Xeon E5-2620 (Pthreads)	11	1458.617	1.349	143.967	58.331	0.761	1.772
Jetson TK1 LP	3	3772.305	0.522	5.628	5.897	0.077	6.779
Jetson TK1 HP	10	1924.222	1.023	9.251	4.944	0.065	15.850
RPi 2 LP	10	11570.112	0.170	2.162	6.948	0.091	1.868
RPi 2 HP	10	11584.317	0.170	2.077	6.684	0.087	1.954



(a) Raspberry Pi 2 (b) Jetson TK1

Figure 7. OpenMP vs. Pthreads - execution times (JTK1 and RPI2).

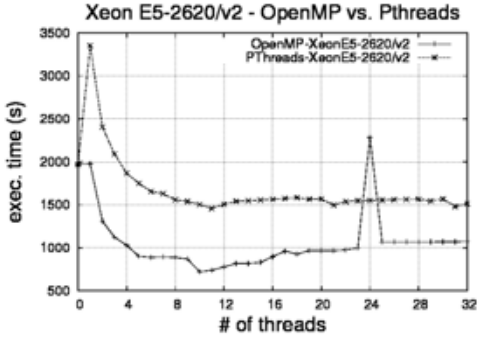


Figure 8.OpenMP vs. Pthreads - execution times (Xeon E5-2620/v2).

5.2.4 OPENMP-MMP VS. PTHREADS-MMP

We now briefly compare the performances of OpenMP-MMP vs. Pthreads-MMP, since both programming paradigms are based on threads. Across all platforms, the OpenMP-based implementation systematically yields higher performance than Pthreads-MMP as can be seen onFigure 7 and Figure 8. This is quite surprising as the used GCC implementation of OpenMP uses Pthreads for its thread operations. We hypothesize that the performance of the OpenMP implementations is partially due to its use of *teams of threads*[31] that work practically in lockstep. This way, all active threads are simultaneously executing the same code and, in the case of OpenMP-MMP, over the same data, thus benefiting from code and data caches. Profiling of the Pthreads-MMP execution on the Xeon server shows that the maximum percentage usage for any

of the (virtual) core never reaches more than 65%, while CPU usage under OpenMP-MMP is close to 100%. This also explains the almost constant average power demand of Pthreads-MMP which is close to 144 watts, regardless of the number of working threads, while OpenMP-MMP requires an average of 167.801 watts when it achieves its fastest execution on the Xeon server. It is important to note that Pthreads-MMP also relies on a pool of threads, and that some further optimization efforts were done to this version, but the performance achieved by OpenMP-MMP could not be matched.

5.3 CUDA

Under CUDA, execution times are dependent on the execution geometry, that is, i) the number of blocks per grid and ii) the number of threads per block[32]. The best geometry configuration, that is, the configuration that yields the fastest execution times, depends not only on the application, but also on the GPU. For MMP, the configurations that yielded the fastest execution times per GPU are shown in Table 8. These best geometries were found through experimentation, although since version 6.5, CUDA provides some API calls reporting on the *occupancy* of a GPU that might help to steer the execution geometry towards optimal performance[33].

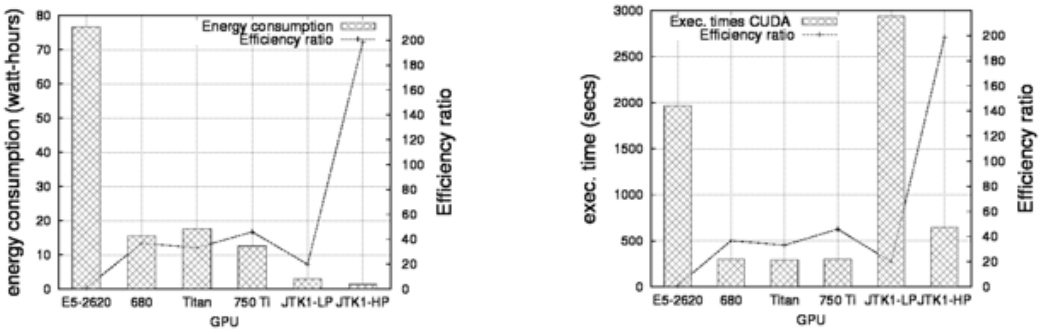
Table 8. Best CUDA geometries per GPU for CUDA-MMP.

	GTX680	GTXTitan Black	GTX750 Ti	JetsonTK1
Blocks	1024	1024	128	6
Threads per block	128	64	288	608

The CUDA-MMP execution results are shown inTable 9. The plots in Figure 9 show the execution times (left plot) and the energy consumption (right plot). Both plots also display the *efficiency ratio*, again considering Sequential-MMP on the Xeon server as reference. The discrete GPUs present the best execution time results, being roughly 6.5 times faster than Sequential-MMP. Interestingly, the marginal execution time differences among the three GPUs, which have different capabilities, is an indication that the performance of CUDA-MMP is not limited by the GPU, but by the CPU. Therefore, since the GTX 750 Ti has the lowest power consumption, it yields the best efficiency ratio for the set of discrete GPUs. Regarding the Jetson TK1, and contrary to what was previously observed with Sequential-MMP, the difference of efficiency between the *low power* and the *high performance* is huge: 17.46 vs. 175.03. This is an indication that the main difference between the *low power* and the *high performance* occurs mostly on the GPU, benefiting the execution of CUDA code. Indeed, as stated before, in *low power* mode the GPU operates at 72 MHz, while in *high performance* it operating frequency is boosted to 852 MHz. Overall, the Jetson TK1, especially in *high performance* mode is the most efficient. Although its execution times are more than the double of the discrete GPUs, it has a much lower power demand, consuming far less energy.

Table 9. Execution times and power usage CUDA-MMP.

CUDA	Exec. Time (seconds)	Speed (ratio)	Avg. Power (watts)	Energy (Wh)	Energy (ratio)	Efficiency (ratio)
Xeon E5-2620/v2 (sequential)	1967.813	1	140.191	76.630	1	1
GTX 680	299.571	6.569	186.844	15.548	0.203	32.360
GTX Titan Black	293.437	6.706	216.223	17.624	0.230	29.157
GTX 750 Ti	298.737	6.581	150.448	12.484	0.163	40.411
Jetson TK1 LP	2939.568	0.669	3.599	2.939	0.038	17.605
Jetson TK1 HP	646.497	3.044	7.422	1.333	0.017	179.059



(a) Execution times (b) Energy consumption

Figure 9. Main results for CUDA-MMP.

6. CONCLUSION AND FUTURE WORK

This paper studies the performance and energy consumption of the MMP encoder application over several hardware platforms using different programming paradigms, namely Pthreads, OpenMP and CUDA, for the parallelization of the MMP encoder. Overall, the CUDA-based implementations yield the fastest executions with speedups around 6.5 over Sequential-MMP on the Xeon server when the discrete GPUs are used. The fact that the relatively modest GTX 750 Ti delivers a performance quasi-identical to the more powerful GTX Titan Black edition is a sign that CUDA-MMP has reached its performance limit at the GPU level.Energy-wise, the Jetson TK1 in high performance mode running CUDA-MMP solely consumes 1/50 of the energy of the referential version, while it still provides a speedup of 3.044. Clearly, for performance-dependent tasks coded to exploit multi- and/or many-core, and run on energy constrained environments, the Jetson Tk1 appears as the best choice. Its successor, the Tegra X1 chip with an 8-core 64-bit ARM CPU and a 256-CUDA core Maxwell GPU might reinforce this trend.Although the Raspberry Pi 2 is clearly not tailored for high performance, it is nonetheless an interesting cheap educational platform, with its 4-core CPU appropriate to explore the multithreading programming paradigms like Pthreads and OpenMP.Additionally, since it has a low power demand, even lower than the Jetson TK1, the Raspberry Pi 2 might be a viable solution on highly power-constrained environments, where a constantly low power demand is favoured relatively to execution speed and the use of a full Linux distribution is needed. This might be the case in a scenario where a solar panel is used to recharge the battery powering the computing system. Note that for the studied application, there were practically no differences between the *low power* and *high performance* mode on the Raspberry Pi 2. Additionally, for extremely power-constrained environments, the Raspberry Pi 2 might be the only viable solutions of the studied hardware, due to its low power demand. Note that for MMP, the performance and energy consumption there were practically no difference between the *low power* and *high performance* mode.

Relatively to the programming paradigms, this work is another example that CUDA can yield good performance even for challenging non-embarrassingly problems such as MMP. For multi-core and/or multi-CPU systems, OpenMP is clearly the first approach to consider, since its simplicity can also deliver good performance. Custom solutions, like directly handling Pthreads should only be considered if OpenMP fails to deliver the needed performance, and as this work has demonstrated, provide no guarantee of achieving better performance than OpenMP. It is also important to note that coding for performance might requires different parallelization approaches depending on the targeted platforms. Considering MMP, its adaptation to multi-core/multi-CPUs required a different parallelization than the one followed for many-core GPUs, namely CUDA.

As future work, we plan to study the performance and energy consumption of MMP versions using the OpenCL paradigm, exploring the advantage of the OpenCL availability for both multi-core/multi-CPU environments, as well as for many-core hardware[2]. We also aim to explore OpenMP with GPUs, taking advantage of existing implementation of version 4 of OpenMP for accelerators[34].

ACKNOWLEDGEMENTS

Financial support provided in the scope of R&D Unit 50008, financed by the applicable financial framework (FCT/MEC through national funds and when applicable co-funded by FEDER - PT2020 partnership agreement). We would like to thank Dr.Pedro Marques and Gilberto Jorge for their valuable contributions with the energy consumption measurements.

REFERENCES

[1] R. Farber, CUDA Application Design and Development, M. Kaufmann, Ed., 2011.
[2] J. E. Stone, D. Gohara e G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," Computing in Science and Engineering, vol. 12, pp. 66-73, 2010.
[3] J. Nickolls, I. Buck, M. Garland e K. Skadron, "Scalable Parallel Programming with CUDA," Queue, vol. 6, n.º 2, pp. 40-53, March 2008.
[4] A. R. Brodtkorb, T. R. Hagen e M. L. Saetra, "Graphics processing unit (GPU) programming strategies and trends in GPU computing," Journal of Parallel and Distributed Computing, vol. 73, pp. 4-13, 2013.
[5] B. Lewis e D. J. Berg, Multithreaded programming with Pthreads, Prentice-Hall, Inc., 1998.
[6] B. Chapman, G. Jost e R. Van Der Pas, Using OpenMP: portable shared memory parallel programming, vol. 10, MIT press, 2008.
[7] S. Wienke, P. Springer, C. Terboven e D. an Mey, "OpenACC-first experiences with real-world applications," em Euro-Par 2012 Parallel Processing, Springer, 2012, pp. 859-870.
[8] NVIDIA, "NVIDIA Tegra K1 - A New Era in Mobile Computing (whitepaper)," NVIDIA Corporation, 2014.
[9] E. Upton e G. Halfacree, Raspberry Pi user guide, John Wiley & Sons, 2014.
[10] P. S. Paolucci, R. Ammendola, A. Biagioni, O. Frezza, F. L. Cicero, A. Lonardo, M. Martinelli, E. Pastorelli, F. Simula e P. Vicini, "Power, Energy and Speed of Embedded and Server Multi-Cores applied to Distributed Simulation of Spiking Neural Networks: ARM in NVIDIA Tegra vs Intel Xeon quad-cores," May 2015.
[11] L. Clarke, I. Glendinning e R. Hempel, "The MPI message passing interface standard," em Programming environments for massively parallel distributed systems, Springer, 1994, pp. 213-218.
[12] G. León, J. Molero, E. Garzón, I. a. P. A. García e E. Quintana-Ortí, "Exploring the performance-power-energy balance of low-power multicore and manycore architectures for anomaly detection in remote sensing," The Journal of Supercomputing, vol. 71, n.º 5, pp. 1893-1906, 2015.
[13] M. Fatica e E. Phillips, "Synthetic Aperture Radar imaging on a CUDA-enabled mobile platform," em High Performance Extreme Computing Conference (HPEC), 2014 IEEE, IEEE, 2014, pp. 1-5.
[14] F. P. Tso, D. R. White, S. Jouet, J. Singer e D. P. Pezaros, "The Glasgow Raspberry Pi cloud: A scale model for cloud computing infrastructures," em Distributed Computing Systems Workshops (ICDCSW), 2013 IEEE 33rd International Conference on, IEEE, 2013, pp. 108-112.
[15] C. Baun, "Performance and energy-efficiency aspects of clusters of single board computers," International Journal of Distributed and Parallel Systems (IJDPS), vol. 7, n.º 2/3/4, pp. 13-22, 2016.
[16] V. W. Lee, C. Kim, J. Chhugani, M. Deisher, D. Kim, A. D. Nguyen, N. Satish, M. Smelyanskiy, S. Chennupaty, P. Hammarlund e others, "Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU," SIGARCH Comput. Archit. News, vol. 38, n.º 3, pp. 451-460, June 2010.
[17] R. Bordawekar, U. Bondhugula e R. Rao, "Believe it or not!: mult-core CPUs can match GPU performance for a FLOP-intensive application!," Proceedings of the 19th international conference on Parallel architectures and compilation techniques, pp. 537-538, 2010.

[18] A. Stamatakis e M. Ott, "Exploiting fine-grained parallelism in the phylogenetic likelihood function with MPI, Pthreads, and OpenMP: a performance study," em Pattern Recognition in Bioinformatics, Springer, 2008, pp. 424-435.
[19] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110," NVIDIA Corporation, 2012.
[20] NVIDIA, "A Multi-core CPU architecture for Low Power and High Performance," NVIDIA Corporation, 2011.
[21] D. Chheda, D. Darde e S. Chitalia, "Smart Projectors using Remote Controlled Raspberry Pi," International Journal of Computer Applications (0975--8887), vol. 82, pp. 6--11, 2013.
[22] L. Dagum e R. Enon, "OpenMP: an industry standard API for shared-memory programming," Computational Science & Engineering, IEEE, vol. 5, pp. 46-55, 1998.
[23] D. R. Butenhof, Programming with POSIX threads, Addison-Wesley Professional, 1997.
[24] C. Nvidia, "NVIDIA CUDA C Best Practices Guide - CUDA Toolkit v7.0," NVIDIA Corporation, 2015.
[25] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer e K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA," Journal of Parallel and Distributed Computing, vol. 68, pp. 1370-1380, 2008.
[26] N. M. Rodrigues, E. A. da Silva, M. B. de Carvalho, S. M. de Faria e V. M. M. da Silva, "On dictionary adaptation for recurrent pattern image coding," Image Processing, IEEE Transactions on, vol. 17, pp. 1640-1653, 2008.
[27] M. B. De Carvalho, E. A. Da Silva e W. A. Finamore, "Multidimensional signal compression using multiscale recurrent patterns," Signal processing, vol. 82, pp. 1559-1580, 2002.
[28] I. E. Richardson, "H264/MPEG-4 Part 10 White Paper-Prediction of Intra Macroblocks," Apr, vol. 30, pp. 1-6, 2003.
[29] G. J. Sullivan, J.-R. Ohm, W.-J. Han e T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," Circuits and Systems for Video Technology, IEEE Transactions on, vol. 22, pp. 1649-1668, 2012.
[30] Microchip, "MCP39F501 datasheet," Microchip Technology Inc, 2013.
[31] D. Novillo, "OpenMP and automatic parallelization in GCC," Proceedings of the GCC Developers Summit, 2006, 2006.
[32] N. Wilt, The CUDA Handbook: A Comprehensive Guide to GPU Programming, Pearson Education, 2013.
[33] NVIDIA, "CUDA Best Practices Guide," Nvidia Corporation, 2015.
[34] A. OpenMP, "OpenMP application program interface version 4.0," 2013.

AUTHORS

Pedro M. M. Pereira received in 2013 his B.Sc. in Computer Engineering from InstitutoPolitécnico de Leiria, Portugal. He is currently a M.Sc. student Mobile Computing at the same institution. His interests include low level algorithms, energy efficient coding in high-performance computing and artificial intelligence.

Patricio Dominguesis with the Department of Informatics Engineering at ESTG-InstitutoPolitécnico de Leiria, Portugal. He holds a Ph.D. (2009) in Informatics Engineering from the University of Coimbra, Portugal. His research interests include multi-core and many-core systems, parallel computing and image, video processing and digital forensics.

Nuno M. M. Rodrigueshas a Ph.D.from Universidade de Coimbra (2009), in collaborationwith Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil. Since 1997, he has been with the Department of Electrical Engineering, ESTG/InstitutoPolitécnico de Leiria, Portugal. His current research interests include image compression and many-core programming.

Gabriel Falcãoholds a Ph.D. degree in electrical and computer engineering from the University of Coimbra, Coimbra, Portugal. He is currently an Assistant Professor at the University of Coimbra and a Researcher at Instituto de Telecomunicações. His research interests include high-performance and parallel computing, hybrid computation on heterogeneous systems and digital signal processing algorithms. Dr.Falcao is a member of the IEEE Signal Processing Society.

Sérgio M. M. de Fariaholds a Ph.D. degree in Electronics and Telecommunications from the University of Essex, England (1996). He is Professor at ESTG/IPLeiria, Portugal, since 1990 and a Senior Researcher with Instituto de Telecomunicações, Portugal. He is an Area Editor of Signal Processing: Image Communication, and a reviewer for several scientific journals and conferences (IEEE, IET and EURASIP). He is a Senior Member of the IEEE.