ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Energy Aware Mapping of a Biologically Accurate Inferior Olive Cell Model on the Single-Chip Cloud Computer

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Γεώργιου Θ.
Χατζηκωνσταντή

Επιβλέπων: Δημήτριος Ι. Σούντρης
Επίκουρος Καθηγητής

Αθήνα, Σεπτέμβριος 2013

Εθνικο Μετσοβιο Πολυτεχνειο
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΜΙΚΡΟΫΠΟΛΟΓΙΣΤΩΝ
ΚΑΙ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Energy Aware Mapping of a Biologically Accurate Inferior Olive Cell Model on the Single-Chip Cloud Computer

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Γεώργιου Θ.
Χατζηκωνσταντή

Επιβλέπων: Δημήτριος Ι. Σούντρης
Επίκουρος Καθηγητής

Εγκρίθηκε από την τριμελή επιτροπή την -ημερομηνία εξέτασης-

.....	.....	.....
Δημήτριος Ι. Σούντρης	Κωνσταντίνα Σ. Νικήτα	Κιαμάλ Ζ. Πεκμεστζή
Επίκουρος Καθηγητής	Καθηγήτρια	Καθηγητής

Αθήνα, Σεπτέμβριος 2013.

.....
Γεώργιος Θ. Χατζηκώνσταντης
Διπλωματούχος Φοιτητής
Εθνικού Μετσόβιου Πολυτεχνείου

Contents

List of Figures

Περίληψη

Το Single-Chip Cloud Computer (SCC) είναι "μια πειραματική πλατφόρμα με 48 πυρήνες Πέντιουμ από την Intel Labs". Με σκοπό την εξερεύνηση των δυνατοτήτων του SCC, δύο εφαρμογές μεταφέρθηκαν στην πλατφόρμα στα πλαίσια της διπλωματικής. Η πρώτη εφαρμογή είναι ένας συμπιεστής εικόνων βάση του πρωτοκόλλου της ομάδας "Joint Photographics Experts Group" (JPEG). Η μεταφορά του συμπιεστή στο SCC πετυχαίνει ικανοποιητική αύξηση της ταχύτητας επί της μονονηματικής έκδοσης της εφαρμογής και προσφέρει έναν αποδεκτο τελικό ρυθμό παραγωγής εικόνων για τις σημερινές ανάγκες της βιομηχανίας.

Η δεύτερη και κύρια εφαρμογή πού μεταφέρθηκε στο SCC είναι ένας προσομοιωτής δικτύων των εγκεφαλικών κυττάρων πυρήνα κάτω ελαίας, με στόχο την ελλάτωση κατανάλωσης ενέργειας λειτουργίας. Καταπιάνεται με την εξερεύνηση του ανθρώπινου εγκεφάλου που απασχολεί μεγάλο μέρος της ακαδημαϊκής έρευνας σήμερα. Διαφοροποιείται από τις συνηθισμένες προσεγγίσεις τύπου "μαύρου κουτιού" καθώς βασίζεται σε ένα βιολογικά ακριβές μοντέλο. Η μεταφορά στο SCC επικεντρώνεται στην εύρεση μεθόδων για την βελτιστοποίηση της απόδοσης του προσομοιωτή σε συνδυασμό με την μείωση του ενεργειακού κόστους της εφαρμογής. Επί τούτου αναπτύχθηκαν δύο μέθοδοι μεταφοράς στο SCC, καθεμία με διαφορετικό τρόπο μείωσης της ενεργειακής κατανάλωσης. Οι δύο μέθοδοι συγκρίνονται βάση γραφημάτων και παρουσιάζεται μια λύση που ισορροπεί ανάμεσα στην ταχύτητα και το καλό ενεργειακό προφίλ της εφαρμογής.

Λέξεις Κλειδιά: SCC, RCCE, HPC, JPEG, Κάτω Ελαία, Ενέργεια, Ισχύς, Εγκέφαλος, Δίκτυο Κυττάρων, Προσομοιωτής

Abstract

The Single-Chip Cloud Computer (SCC) is an experimental board with 48 Pentium cores created by Intel Labs. To explore SCC, this thesis covers the porting of two applications on the board. The first project is the porting of an encoder for the "Joint Photographics Experts Group (JPEG) Protocol for still image compression". Porting the encoder on the SCC yields a satisfactory speedup of the single-threaded version and achieves an acceptable frame rate with respect to today's standards of the industry.

The second and main application ported on the SCC is an energy-aware simulator of inferior olive cell networks. It tackles an important aspect of the human brain's exploration, a subject motivating academic research greatly in modern times. It differs from the usual black-box approach on the matter by using a biologically accurate model. The porting focuses on finding efficient solutions to optimizing the simulator's performance while reducing energy expenditure and power consumption. To this end, two different porting options are introduced, each with a different method of lowering power requirements. The different methods are compared against each other with extensive Figures detailing each option's results. Ultimately, a solution that balances performance and energy gain is presented.

Keywords: SCC, RCCE, HPC, JPEG, Inferior Olive, Energy, Power, Brain, Cell Network, Simulator

Acknowledgements

The following thesis is the fruit of my work on the SCC Platform over a period of one year (fall of 2012 up to fall of 2013) and my collaboration with the Microprocessors and Digital Systems Laboratory of NTUA. I would like to thank my supervisor, Prof. Dimitrios Soudris for the opportunity of working on the SCC and his guidance, as well as Mr. Dimitrios Rodopoulos for supporting me and greatly helping me tackle the board's problems and complexities, while providing valuable insight throughout this thesis. Credits are due to our partners in Erasmus University of Rotterdam, who supplied us with the baseline of the Inferior Olive Simulator and helped with its porting on the SCC and its fine-tuning. Special thanks go to Dr. Christos Strydis, who also acted as the intermediary between NTUA and Erasmus.

My long journey leading to graduating from NTUA would not be possible without the guidance of my experienced Professors. I would especially like to thank my Professor Dr. Georgios I. Goumas for his excellent lectures on Parallel Processing Systems and Professor Nectarios Koziris for his lessons on Operating Systems and Computer Architecture. Both Professors inspired me greatly to study computer science in more detail.

I want to thank fellow students and friends that have helped me tackle the challenges NTUA presents its students. Special mention goes to Ira Ktena, Panagiotis Traganitis, Giannis Papastathis and Konstantinos Gkinis, all of whom greatly assisted me during all these years. Finally, I feel the need to thank my parents for their endless support since my birth.

Chapter 1

Introduction

The SCC experimental processor [27] is a 48-core "concept vehicle" created by Intel Labs as a platform for many-core software research.

This thesis covers the mapping of two applications on the SCC. The first application is an encoder of the JPEG format. The JPEG encoder served as an application to explore the capabilities of the SCC and familiarize the user with its programming paradigm. Image compression and processing is a subject of great value in today's era. Many Computer Generated Image (CGI) studios demand ever increasing processing power and revolutionary progress is marked in the field of medical imaging. Thus, a JPEG encoder served as a representative concept vehicle to create a first functional project on the SCC.

The second application mapped on the SCC is a biologically accurate simulator of the inferior olive cell activity. It constitutes the main contribution of this thesis and involves an exploration of mapping options, along with an evaluation of the associated quality costs (energy, delay etc). Contrary to most existing approaches on cell activity modeling (black box approaches [37]), this simulator is based on mathematical equations concerning cells' channel conductances, in order to calculate important parameters of the cells, such as its compartments' voltage levels. Because of the accuracy of the utilized model, it is a great tool to observe the obscure behavior of important brain cells when exposed to user-defined stimuli. The sheer volume of floating point operations during a simulation makes the application suitable for speeding up on the 48 cores of the SCC. This is done via both data and task partitioning; these two main techniques point to how workload can be distributed across the available cores. Data partitioning refers to segmenting and assigning different parts of the program's data to different cores and performing the same actions on them. Task partitioning on the other hand, refers to assigning different tasks to different cores and each one performing on the entirety of the application's data. Both techniques have their merits and the best multi-core performance possible is often attained by utilizing a mix of both methods [29].

This thesis holds an overview of the SCC platform in Chapter 2. Information about Intel's

multi-core project and SCC's history is provided, followed by presentation of the chip's architecture. The board's programming paradigm is then detailed, along with libraries and utilities offered to the user to fulfill his needs and ease his familiarization phase with the SCC. Special attention will be given to power management utilites, which are extensivelly used in the inferior olive simulator's design. Chapter ends with the remarks about the user's experience working with the board.

Chapter 3 and Chapter 4 detail the porting of the JPEG encoder and the inferior olive simulator on the SCC, respectively. Both Chapters begin by describing the nature of each application's purpose and what motivated their development. Details about both applications' structure are provided. In the case of the inferior olive simulator of Chapter 4, two separate porting options are presented along with two methods aimed at lowering energy consumption while maintaining peak performance. Results are discussed based on extensive Figures describing variables examined, such as frame rate output for the JPEG encoder, simulation completion time and energy expenditure for the inferior olive simulator etc.

The thesis concludes in Chapter 5 which summarizes both applications' performance. Important conclusions valuable to the reader are discussed. Suggestions for improving the existing work are mainly held in this Chapter. The Chapter ends with a remark from the author on how the SCC sets the trend for future developers caring for the energy profile of their applications, as well as providing them the chance to develop multi-core projects, ultimately succeeding in its original purpose.

Chapter 2

SCC Platform Overview

2.1 Introduction

The SCC is a platform developed by Intel corporation largely for research on multi-core programming and architecure. The platform proceeded a previous 80-core platform named Teraflops Research Chip (also named Polaris [31]). Both are part of Intel's Tera-scale Project [30]. The project's aim is to answer various important questions concerning highly parallel computing (HPC), such as what programmers can achieve when they are equipped with software tools enabling control over power consumption of applications and how scalable networks of cores need to be interconnected. According to Intel Labs, "the SCC is an ideal research platform to help accelerate many-core software research" [16].

This Chapter will initially describe the architecture of the SCC and its necessary connection to the host Personal Computer (host PC). Then the process of programming will be described, as well as the available software tools to write multi-core, communication-heavy applications. Finally, any additional utilities provided by Intel Labs that have been proven useful will be relayed.

2.2 Physical Description

In its current form, the SCC needs a host PC called the Management Console PC (MCPC) which connects to the board via PCI-Express bus. The MCPC runs a Linux distribution of the user's choice, while the cores have the option of supporting baremetal applications or loading Linux images [15]. The thesis has only tackled applications using a working Linux OS loaded on the cores and thus will not include further information concerning baremetal applications.

The board itself consists of 48 cores on a 6×4 mesh. The mesh is split in 24 tiles containing 2 cores each. The tile is the basis for the SCC, each one being an autonomous entity, connected with the rest of the network of tiles. For each core, there is a separate L1 Instruction and a L1 Data cache, while the L2 cache is common for both and is located on the tile. Both L1 caches' capacity is 16KB, while the L2 cache is 256KB. On the tile, a crucial part of core communication is located, the message passing buffer (MPB). The MPB is a small SRAM buffer of 16KB and its contents can be read by any core of all 24 tiles. In total, there is 384KB of message-passing memory that can be accessed by any core on the SCC. Hence, whenever a message needs to be sent from one core to the other, the cores transfer the required data from their L1 cache to their MPB, so that the receiving core can access the information and copy it to its own L1 cache.

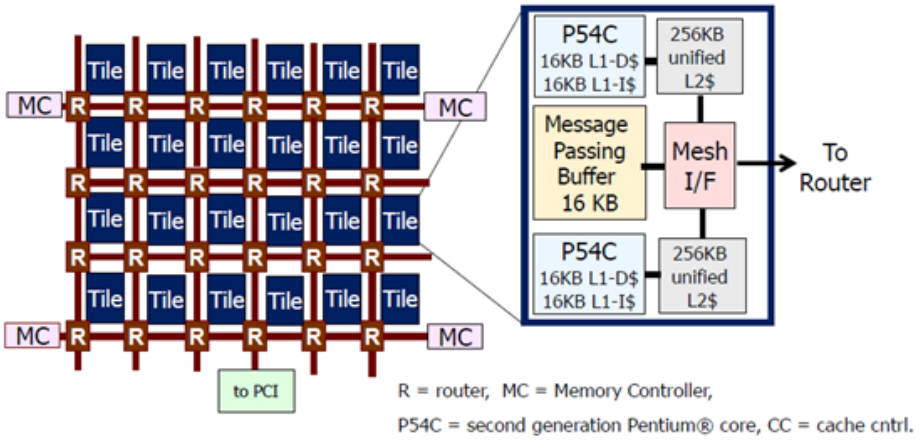

Figure 2.1: A brief description of the SCC Architecture [30]

"The Mesh Interface Unit (called MIU) connects the tile to the mesh. It packetizes data out to the mesh and unpacks data in from the mesh" [16]. Because the core addresses are 32-bit but the SCC holds 64GB of RAM, the MIU acts as a decoder of a core address to a system address. This process uses a look up table for translating, the table being different for each core. The MIU also handles the data flow on the mesh. A credit-based protocol regulates data traffic incoming to or outgoing from the tile. The flow alternates between the two cores of the tile based on a round robin protocol.

As mentioned before, in the configuration used for this research, the SCC was connected to a host PC, the MCPC through a PCIe bus. The MCPC is a 64-bit console that runs Linux,

with all required software for programming provided by Intel. Through such software, the SCC platform is configured and then the desired application is compiled and loaded on the SCC cores. All Input/Output (I/O) operations of the application are carried out through the MCPC. In this configuration, the files and logs that serve as I/O for the application that is executed on the SCC are kept in a special folder on the MCPC that maps to a shared memory space between the console and the SCC. Of the board's 64GB RAM, 32GB is private to the cores and the other 32GB is shared between the SCC and the MCPC. All files in that space (and only those) can be created by the MCPC and be accessible to the cores. Thus, they are used for the required input (list of frames and frames-to-be-encoded for the JPEG encoder, simulation parameters specificiations for the inferior olive simulator) as well as output (encoded frames and logs detailing information about the JPEG encoder's performance, inferior olive simulator output files) of both applications mapped on the SCC in this thesis.

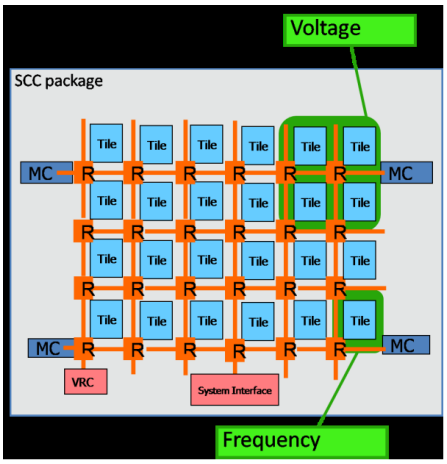

Figure 2.2: Voltage and Frequency Domains on the SCC Mesh [16]

Other important compartments of the SCC board include a power controller. This controller allows the user to alter the frequency and voltage under which a subset of cores operates. A program can affect in real time the frequency level of a tile and the user can also affect the voltage level of a mesh of 4 tiles (2×2, 8 tiles total), as shown in Figure 2.2. However voltage adjustment takes considerable time and cannot be realistically achieved on the fly. The board also contains a digital temperature sensor which allows observation of the tiles' cores, even as their frequency and voltage feed is altered. These sensors are available via the `sccTherm` program [35].

2.3 Programming Model

The application developer mostly works on the MCPC. All necessary utilities are installed there, the cores of the SCC have minimal user interface and can even function without an OS (baremetal applications). The MCPC is a typical PC with Linux distribution, facilitating the developer as he works in a familiar environment.

For the programming of many-core applications demanding communication between cores, a many-core C-language environment very similar to the open-mpi library [10] is used and provided by Intel, called **RCCE** [15] [22]. Its purpose serves writing message passing application programs on the board. The RCCE library includes a vast range of functions, from simple message passing between two cores to broadcasting messages from one core to the entire mesh. One who is familiar with the logic of message passing parallel programming is quick to adapt to and use the RCCE environment.

The RCCE library has two main modes: gory mode and non-gory mode. The difference between the two is the amount of detail the programmer gets involved with concerning message passing. Non-gory mode is usually enough for an average application and was the mode selected when the library was first configured and built for the SCC system. Gory mode was not explored at all for this thesis.

Most applications, including those developed in this thesis, make use of `RCCE_send()` [15] and `RCCE_recv()` functions. These are simple blocking functions transmitting a buffer of predefined size between two cores. Like any blocking communication function, each `RCCE_send()` must be met with a corresponding `RCCE_recv()` from the core receiving the information and vice versa. There are non-blocking versions of the functions developed by RWTH Aachen University [24] which were not used by the applications presented in this thesis.

There are more collective communication functions enabling mass core-to-core data exchange. `RCCE_scatter()` [22] and `RCCE_gather()` will be mentioned here, which allow a core to send the contents of a buffer to multiple other cores silmutaneously, as well as receive information from multiple cores. Such functions perform better than individual send-receives since they are optimized for mass communication and they prevent deadlocks; however they can only be employed with careful processing of data-to-be-sent and received. They also demand communication between all cores employed by the application, which is not always desirable.

The usual process of programming on the SCC is as follows: the developer works on his project on the MCPC in a private folder. He then compiles the project with the necessary adjustments to a regular makefile so that the RCCE library is included and the executable is compatible for the P54C architecture of the SCC cores. Then the executable is transferred to the shared folder with the board, along with a file describing the core addresses that will be demanded to host the application, as well as any input and output files necessary to the program. A provided script is then initiated which forces the cores to execute the application.

If the execution runs into a critical error, an error message along with an error code is returned to the MCPC by each core of the SCC individually. The developer can look up the error code to discern problems occuring in his project leading to failure [17].

2.4 Utilities and Experiences

The user is provided with a script (named `rccerun`) to facilitate dispatching jobs to the SCC. The script first flushes the MPBs of the cores requested by the user to host the program. It then proceeds with the executable, assigning it to the cores as indicated by the host file provided by the user. The script is important, as the MPBs are necessary for any communication between cores and previous programs may leave them in undefined state.

Intel Labs provides the user with software for the MCPC called `sccKit` [16]. `sccKit` offers a graphic user interface (`sccGUI`) to supervise the condition under which the cores operate. `sccGUI` contains information on how much of each core's processing power is being used. It also helps at the startup of the cores, allowing easy booting of Linux on all cores. `sccKit` also offers the `sccKonsole` command, allowing the user to access a terminal on any of the 48 SCC cores. The protocol used to connect to the cores via the MCPC is SSH [7]. If the user wants to execute a particular command in all core terminals (the top command for example), there is a broadcast option to copy input from one terminal to all others.

A factor that draws people to the SCC would definitely be its power management utilities [18]. As mentioned in Figure 2.2 of Section 2.2, the board's power controller allows the user to dictate the voltage and frequency of operation of the cores, within certain limitations. This allows the developer to experiment with energy consumption reduction methods, an issue that is very pressing and interesting these days [40], as he can observe how the application behaves when running under different frequency levels. It also allows to balance workload "slack" between cores handling different tasks with energy gains by allowing these particular cores to operate at lower voltage settings.

One important asset of the SCC, which renders it suitable for experimental purposes, would be how familiar to the user its programming paradigm is. The board itself does not burden the cores with anything unnecessary, leaving it up to the user to boot an operating system, if any at all. The MCPC on the other hand is a typical Linux system, equipped with all necessary compilers for the P54C architecure, code editors and general utilities the programmer needs, such as internet browsers. This way, while the SCC is minimalistic, it offers the developer with everything required to develop his project in a friendly environment.

There is a great variety of sources from which the programmer can acquire information concerning both the SCC and the RCCE library. Many manuals concerning the board and its usage have been composed by Intel which have been invaluable to this project (some of them are used as references for this thesis). There is also a very active Intel forum helping out and educating SCC developers [2]. Finally, many of the projects already created by third parties for the SCC, such as the iRCCE library [24], are open to the public, allowing a developer to collect ideas, inspiration and acquire information.

The SCC presents some troubles with debugging. Any messages the cores try to send to the

MCPC seem to fail when the program freezes for whatever reason. Thus a simple segmentation fault causing the execution to crush might not be caught by the SCC if it causes the program to "hang" and no error code is returned to the user. Also, any messages embedded within the application to help the developer keep track of his project are delayed, so if the program crushes, they never appear to help the user know which part of the code failed. Furthermore, when a program freezes for any reason while executed by the core, the easiest way to terminate it is to run the provided script killing a particular set of jobs on all cores, as defined by the user. These problems cannot be practically alleviated by the use of a debugger either, since an application running on multiple cores would require an instance of the debugger running on each of the cores utilized by the application. All these minor details combined together make the process of debugging somewhat tedious for the developer, forcing him to waste development time over trivial errors in his application.

To summarize, the experience of the author using the SCC has been overall positive. There is plenty of support for the developer provided by Intel, such as technical reports and hardware or software manuals, along with a dedicated forum [2]. The MCPC, once configured correctly, provides everything one needs to begin creating multi-core applications. This is done in an environment that is familiar to the programmer, with tools that resemble well-known pre-existing ones, like then open-mpi library [10]. While there are some setbacks during the development phase of an application due to troublesome bug handling, the SCC is deemed an excellent advanced platform to compose and test HPC projects of any nature. This is further supported by the fact that future products follow, like the SCC did, the many-core paradigm, such as the "Intel Xeon Phi coprocessor" [34] [8] and the ST Microelectronics P2012 "area- and power-efficient many-core computing fabric" [36].

Chapter 3

Porting of the JPEG Encoder

3.1 Introduction

The first portion of the thesis on the Intel SCC is the porting of a JPEG protocol encoder. The application has the following task in mind: given a set of `.bmp` format pictures, compress the images and transform them in JPEG images, in the least amount of time possible.

The name JPEG stands for "Joint Photographic Experts Group", the name of the committee that created the JPEG standard, as well as other still picture coding standards. The "Joint" in JPEG refers to the collaboration of two groups, CCITT and ISO [9]. It is the most common method of lossy compression for digital photography. Particularly, all contemporary digital photo cameras support the capture of images in JPEG format as the common medium for image interchange and all image viewers, image editors and Web browsers can display JPEG images as a common standard. It was first publicly released in October 1991 and has been developed since that time. In June 2009, JPEG Group published version 7 of the software with new features for image coding application. In January 2010, version 8 was introduced with extension providing the basis for the "next generation image coding standard" [5]. The current version is release 9 of 13-Jan-2013.

JPEG's goal has been to develop a method for image compression which meets a number of criteria [46]. The standard needs to be at or near state-of-the-art with regard to compression rate and accompanying image fidelity, over a wide range of image resolutions, sizes and quality ratings. The application also needs to be configurable so that the user can set the compression/quality ratio which is desired. The standard needs to be applicable to any kind of digital-source image and not be restricted by dimension, color, aspect ratios or any other parameter. It also needs to have a tractable computational complexity so as implementation is feasible to program. Lastly, it needs to have different modes of operation, sequential encoding, progressive encoding, lossless encoding and hierarchical encoding.

In the context of this thesis, the JPEG encoder was used as a concept vehicle to attain

familiarization with the SCC architecture and its programming paradigm, the RCCE library. As such, in order to achieve familiarity and gain experience with it, an application that could easily achieve high degree of parallelism was deemed necessary. An encoder is such an application, demanding high levels of throughput, encoding at a fast pace multiple frames at high resolutions so as to achieve high frame-per-second (FPS) delivery times.

However, it is worthy of note that instead of simply assigning different images to different cores with minimal effort, the encoding algorithm was broken down in stages and a pipeline architecture was realized to exploit task partitioning while processing a single image. The advantages of such an approach, discussed in detail further below, include specialization of the tasks each core undertakes and providing the user with the option of assigning more computational resources to process images of greater priority and/or importance.

3.2 Motivation

Over the years, the need for great computing power to serve the needs of image processing has increased. Tasks, such as rendering large objects of high resolution, which have been considered impossible in the past can be achieved today with high-performance computing [39]. As a result companies interested in the field of image rendering and animation invest in obtaining higher levels of processing power. An example of this would be the collaboration of Intel Labs and Pixar, the well-known animation studios [20]. As time progresses, more and more processing power is required to the development of CGI-based films. Rendering, which is the process of creating an image on the computer from a strict model detailing the geometry, shading, texture and lighting of a three-dimensional object, is a very demanding process, from a computational standpoint. An average film of 90 minutes at 24 frames per second requires the production of over 130,000 frames. However, the task can be handled by efficient parallel programming, as multiple frames can be processed simultaneously. Pixar Studios invest in systems with great amounts of cores, providing them with the computing power they need, as well as upgrading their single-threaded rendering engine (called RenderMan) to supporting multiple threads so as to be able to take advantage of multicore systems provided by Intel.

Highly parallel computing is not only beneficial to image rendering for entertaining purposes. 3D medical information obtained from Magnetic Resonance Image (MRI), X-ray Computed Tomography (CT), etc. are used for an operation-supporting image, and operations under image guidance has also been conducted [19]. Taking it one step further, Integral Videography (IV) can reproduce a computer-generated spatial object. However, due to the sheer amount of calculation needed, parallel processing is paramount to reduce delay in rendering.

Another source of motivation for this project has been a very relevant work from the University of Eindhoven in collaboration with the University of Las Palmas, named MiniNOC [14]. The project focused on using a Network-on-a-Chip to map multi-core applications of vast computational needs. A lot of software tools were developed by the two universities but most importantly for this thesis, a JPEG decoder was ported on the MiniNOC. The results of the work were encouraging and it should be noted that future work includes increasing the speed of the decoding process. Compared to the 2×2-core MiniNOC platform, SCC was deemed a platform capable of better performance. Thus, there have already been attempts at porting the widely used JPEG protocol on a Network on Chip (NoC).

The need for high performance computing (HPC) techniques in the field of image rendering has been proven on many different occasions. There is a general increasing need for HPC in a diverse number of fields, ranging from medicinal [4] and geometrical applications to financial [6] projects. The science of parallel computing is no longer restricted to supercomputing centers as more and more companies invest in HPC. With this in mind, porting an image-rendering-relevant application on a high performance NoC is worthy of taking a look at.

3.3 JPEG Encoder and Porting

There is currently a great number of applications on the field of digital imaging. The key obstacle for many of them is the vast amount of data required to represent a digital image directly. A single, color picture at TV resolution requires megabytes of data to be represented. This amount of data even grows at a fast rate with the increase in image size and resolution.

The JPEG encoder is broken down on separate tasks, feeding data from one task to the next like a streaming application. The procedure will be detailed in full further in the text, however this fact has encouraged the porting of the encoder on a multicore platform for distributing the tasks to cores, in a pipeline fashion. The idea was inspired originally by a number of other projects. One such work, the MiniNOC project has already been mentioned; another is the work of Rodopoulos et al. as presented at the IEEE SELSE workshop in 2012 [41]. This work focused on the application of decoding as a means of exploring silicon error handling protocols. It provided a solid background to work on.

The JPEG protocol is split in 3 stages, as depicted in Figure 3.1. The `.bmp` image is split in 8x8 (pixels) blocks. In the first stage, the block is parsed and decoded in YGB buffers which are fed as input to the next stage. In stage 2, from YGB encoding, through a mathematical process called Discrete Cosine Transform (DCT), the picture is coded as frequency blocks, which are 64-point discrete signals. The frequency blocks are forwarded to stage 3, which quantizes the signal based on a hardcoded frequency table, rearranges the blocks in a zig-zag fashion and further compresses the image based on Huffman's coding.

The blocks are provided as input in stage 1, make their way through the 3 stages and are ready to be recorded in the output `.jpg` file. After every block has been processed, the footer is signed and the `.jpg` file is ready. Thus a pipeline approach has been used so as to increase the encoding throughput. A core is assigned to executing a specific stage. After a block is finished, it is passed on to the appropriate core/stage and the next block is processed. Thus the 48 cores of the SCC are organized in triads in this work, where each core of the triad always handles one particular stage for all blocks of the frames assigned to the triad.

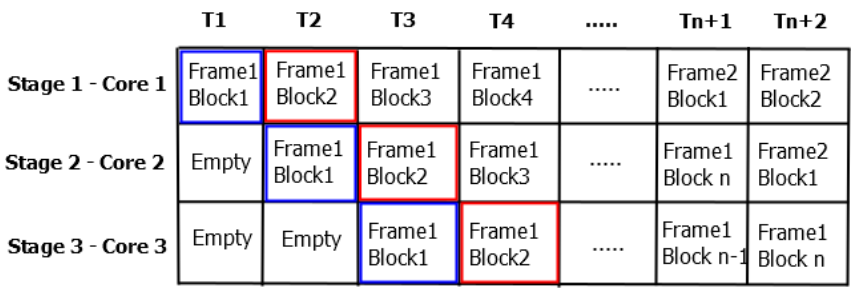

Figure 3.1: Illustration of the encoder pipeline: after n+2 steps, the n-block frame 1 is produced.

One main advantage that should be noted in the particular porting option that was followed is the task partitioning between cores. The DCT transformation is a computationally heavy task, whereas splitting the image in 8x8 blocks and converting them to YGB standard mostly comprises of many memory accesses. Hence, one could assign the former task to a core with strong ALU and the latter task to a core with a fast and suitably sized series of data caches. Specializing cores according to tasks assigned is a profitable method. It can lead to speedup as well as financial benefits, since the user knows exact specifications to build his system with and run the encoder application optimally. Moreover, cores with lighter tasks than others can be tuned to operate at lower frequencies in order to align better with the other cores and save up on energy costs. Such arguments prove the worth of a task-partitioning porting option over strictly data-partitioning methods.

3.4 Results

Several metrics show the efficiency of the application. One important outcome of the application is the frames-per-second (FPS) metric, which shows how rate at which the series of images supplied to the application are being processed and compressed. Figure 3.2 describes how FPS scales as more resources of the SCC are used. It is important to note that because of the particular porting option, the amount of cores used is always a multiple of 3, since each core handles one step of the 3-stage pipeline. A linear increase in FPS is observed as more resources are used, capping at about 65 FPS when the SCC is used at its fullest. The linearity is expected since the application breaks down the amount of images each triad of cores handles according to how many cores are available. What is more interesting is the 48-core result of 65 FPS which is an acceptable benchmark. It is actually more than triple the upper threshold at which the human eye can perceive changes in alternating images and much greater than double the frame rate at which cinematic films have been traditionally shot (24 fps [3], albeit usually at higher resolutions [12] than the small picture used as benchmark for this thesis). Although the size of the image was small, the rate at which it was processed is more than satisfactory and a stronger platform with more cores could easily process higher resolution images even faster.

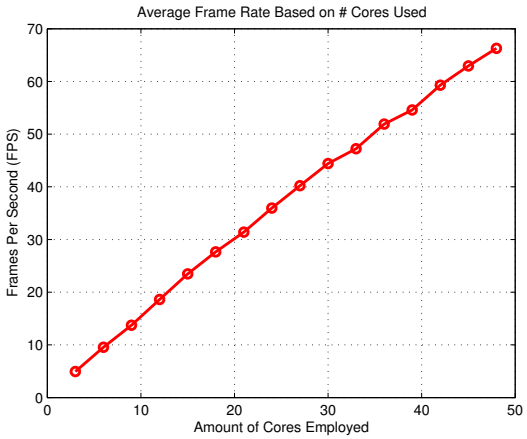

Figure 3.2: Figure detailing Frame Rate scaling with number of cores used

Figure 3.3 details the performance of the application when using the SCC at full capacity. Using as reference point the moment the first frame is produced, the exact time at which the following frames are delivered is recorded. It is observed that at the beginning of the application, the frames are delivered 16 at a time. Since this metric uses all 48 of the SCC cores, equaling to 16 triads of cores, 16 frames are being processed at any given moment. Thus, at the beginning 16 frames are delivered almost concurrently. However, as more images are being processed, the delivery chronograph becomes more noisy. Many reasons can cause individual cores to stall and delay the delivery of a frame(e.g. interrupting requests by the operating system), thus causing the de-syncing of the core triads. This also leads to the delivery of subsequent frames earlier

than current frames. Potentially this could be harmful for an application using the encoder for motion picture purposes (since the frames need to be sorted) and it could be interesting to design a system reallocating frame processing in response to unavoidable and unpredictable stalls. Such factors affect performance variability, an issue intimately linked to many-core systems [21].

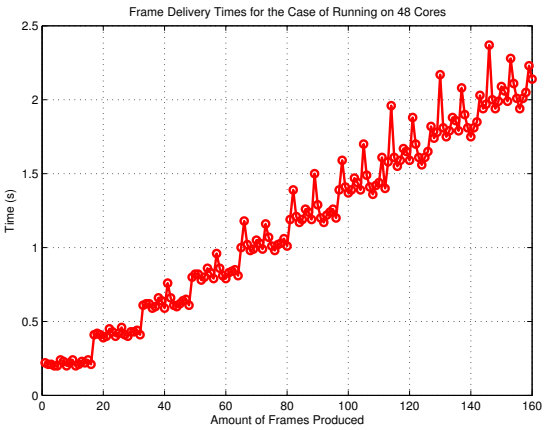

Figure 3.3: Delivery times of compressed frames

Figure 3.4 touches on the subject of power consumption during the benchmark of encoding 160 frames. Power efficiency is an increasing concern for applications running days, months even, causing considerable expenses to companies using them due to energy consumption. The encoder application causes a power pulse for the SCC. Its cores normally operate at 70 Watts, however since the application uses the CPUs to their fullest, power increases for the duration of the encoding. To calculate energy costs, the average power level increase during the spike was calculated and multiplied with the duration of the pulse. Although this method is simple, the SCC does not provide the user with accurate tools. The power level can only be measured on board-level, not core-level (or at least tile-level). When one wishes to measure energy expenditure of an application running on specific cores, measurements become diluted with the power levels of idle cores. Thus, measuring power levels when employing a small amount of cores for the application is inaccurate. One can understand from Figure 3.4 that increasing the amount of cores employed results in greater energy consumption, but it would be of interest to have more accurate graphs and the ability to measure power pulses in cores operating different tasks.

In Figure 3.5, one needs to notice the delay in the fall of the power level after the application has ended. According to Figure 3.3, the application lasts less than 3 seconds when operating with 48 cores, but the power pulse has a width of more than 6 seconds. This observation can be attributed to the inaccuracy of measuring power levels, or an actual delay due to the SCC being unable to adjust its power level swiftly enough.

The results clearly point to the following rule: using an increasing amount of the platform's resources leads to greater speedup and higher power consumption. The user needs to determine

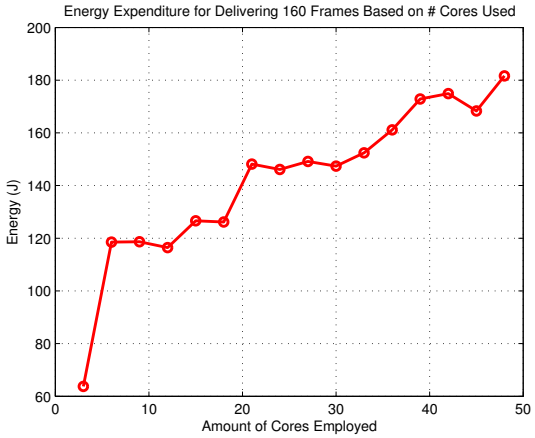

Figure 3.4: Increasing energy consumption cost relevant to amount of cores used

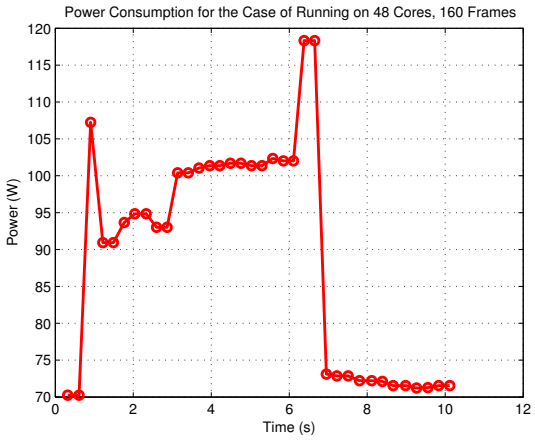

Figure 3.5: Sample power pulse when using the SCC at its full capacity

the frame rate he aims for and use the appropriate amount of cores in order to avoid excess expenditures. However, more accurate measurements on different platforms could produce useful Pareto curves [38] between power consumption and execution speed with varying amount of cores used.

3.5 Future Work and Conclusions

There was a number of different approaches that were examined and might prove worthy of interest for future work. In the encoding process, the blocks in which the image is split are independent from one another. Another, simpler approach would be to assign the entire encoding process to every core of the SCC and distribute different blocks of a single image to different cores. While this approach would lead to good workload distribution between cores (since every core performs the same series of tasks), it would lead to the problem of needing a "master core" responsible for collecting the blocks from the other "slave cores", sorting them and assembling the unified `.jpg` file. This approach effectively "wastes" one core and imposes a bottleneck as one core has to communicate silmutaneously with the other cores (such an approach would be an interesting opportunity of using the scatter and gather functions of the RCCE library [22]).

Another approach would be to assign the encoding of entire images to a single core and map images to cores. However this option can introduce significant delay in the production of the first in a series of images, as well as not providing speedups for the encoding of single images. It would also be a problematic approach in case the images are dependent on one another, such as the case of MPEG protocols for compressed moving pictures [44]. If however this was not the case (as in the Motion-JPEG protocol [11], where each video frame is independently compressed and transmitted in series) then one could very well explore this porting option.

Other than alternate porting options, one of the first aspects that need to be worked on for the class of image processing on the SCC would be increasing its compatibility with images of other protocols (like raster coding). Because of the variety in image protocols, constructing an application able to process image files of different structure is a non-trivial task. This would enable the exploration of whether image format affects the encoder's performace. An easier task would be to make the application compatible with the raster format since it does not have multiple versions, allowing the encoder to be compatible with more images.

Another course of future work is testing the encoder with images of different resolutions. At the moment, the encoder has only been tested with an image of low resolution. FPS output would certainly be lower for images of greater size. It would be interesting to explore how FPS is affected by increasing workload via higher resolution images. Furthermore, large size images are more indicative of today's ever-increasing need for processing great amounts of data.

Conclusions can be drawn from the successful porting of the application. For small images, when operating at the SCC's full processing capacity, a very satisfactory FPS output can be reached. Since increasing the amount of cores employed by the application leads to a linear increase of FPS but a sub-linear increase in power consumption, using higher amount of cores yields a better performance to energy cost ratio.

Chapter 4

Porting of the Inferior Olive Simulator

4.1 Introduction

The main application that has been ported on the SCC for this thesis is a simulator of an important set of brain cells, the inferior olive (IO) cells. The IO cells have been associated with brain functions such as "the learning and timing of movements" [25]. The cells are the receivers of stimuli through the human senses and pass on such input to the cerebellar cortex via the Purkinje cells. Should the IO cells be damaged, the patient is unable to synchronize his movements and thus severe cases of ataxia can be demonstrated [26]. As such, they are an important part of the human brain, which motivated the development of the simulator.

The application aims at simulating a user-defined network of IO cells. This network is provided with a custom input current at each simulation step. Then, the biological parameters of the entire network are calculated and recorded. The application thus calculates and records the voltage levels of each cell as it reacts to the stimuli (input current) as well as other parameters that define its state. As such, the application is biologically accurate and different from most black-box approaches on the matter (e.g. neural networks [37]).

The exploration of the brain's obscure and complex functions is the goal of many large-scale projects today [1]. Great benefits can come out of such studies. One such instance is the development of implantable chips that help patients combat grave diseases such as epilepsy and Parkinson's disease [47]. Other biological studies have lead to breakthroughs in creating artificial pancreas systems which help diabetes patients regulate glycemic levels [23]. The biomedical science is indeed a very interesting and rapidly growing field.

The application also tackles important questions in the field of HPC. It features two different methods of utilizing the SCC's many cores. Porting option #1 is based strictly on data parti-

tioning, while porting option #2 incorporates data and task partitioning. Both options show great speedup in comparison to the single-threaded version of the application, which serves as a reference for comparison [32], as well as different benefits and drawbacks. Apart from different ways of partitioning computational workload, the current thesis also explores the board's utilities for voltage and frequency scaling. Two different methods of reducing power consumption are introduced, a static and a dynamic solution, which are applied to the two alternative porting options. Thus, "the concern of improving power efficiency" [42] in multicore applications is also considered.

After detailing the biological model used for this application and the challenges of improving the single-threaded version, the two porting options will be presented as well as the different methods used for each option, aimed at lessening power consumption. The results of each effort are analyzed in extensive graphs and commented upon. Finally, the author's insight into the different mapping options is presented, as well as possible future work for the simulator.

4.2 Biological and Simulation Model

The IO cell in this simulation is based on a compartmental model. Each cell comprises of 3 compartments: the dendrite, the soma and the axon. Each compartment serves a different purpose biologically and has different membrane voltage levels.

- The dendrite compartment is responsible for communicating with the rest of the cell grid. It has been proven that brain cell networks show a great degree of interconnectivity based on various parameters such as brain size [33]. The dendritic compartment handles communication with other IO cells, which is simulated by the application via recording other dendrites' membrane voltage levels. These levels, along with other biological parameters, greatly affect computations which calculate the dendrite's potential in each simulation step. This compartment also receives stimuli from the environment as input current.

- The somatic compartment is the center of computations for the cell. The most elaborate and time-consuming calculations are handled by the somatic compartment, which also communicates with the other two compartments via voltage levels.

- The axonal compartment is the "output port" of the cell, the compartment whose voltage level is recorded in each simulation step. It features the lightest computational and communication workload, however it performs a lot of I/O operations.

A brief description of each simulation step follows: the application allocates enough space in memory for holding all of the necessary parameters for each cell, such as membrane voltage level for each compartment, various ion concentration levels and communicating cells' dendrite voltage levels. Initial values are randomly given to these biological parameters and a random amount of closed-circuit simulation steps are executed for each cell individually. Thus, randomization of the grid's initial state is achieved before proceeding to the actual simulation.

The dendrite is fed input current in each simulation step as stimuli from the cell network's environment. This can be achieved either by a user-defined input file which details each cell's input for each step, or by a hard-coded spike input current. The second method of input was largely used for debugging purposes as well as porting assessment. The dendrite then records the dendritic voltage levels of its communicating cells. Intercommunication in the network is described by another user-defined file which details incoming and outgoing connections for each cell. Each dendrite needs the voltage levels of each incoming dendritic connection and sends its own voltage level to every outgoing connection. A naive interconnectivity system is simulated via the simpler assumption that each cell communicates only with its neighboring cells in an 8 way connectivity (i.e. the network is represented as a two-dimensional matrix where each element is immediately adjacent to a maximum of 8 other elements). It must be noted however that inter-core communication overhead is greatly lessened with this assumption.

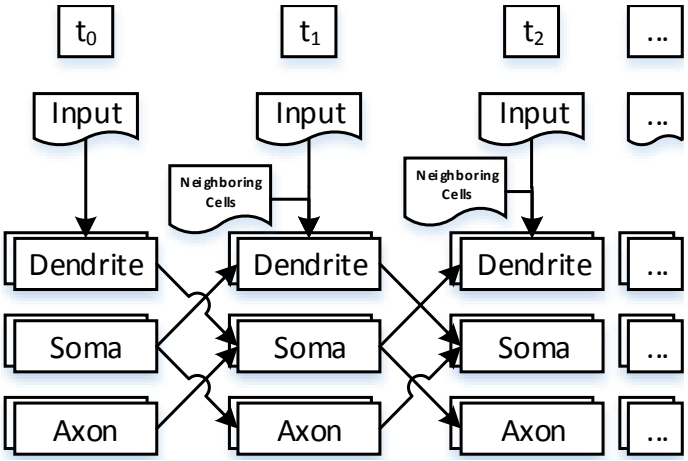

Figure 4.1: Brief presentation of dataflow between simulation steps

After the dendritic compartment has finished collecting information from both outside environment and connecting cells, the compartments share their voltage levels. Specifically, dendritic and axonal compartments need the somatic compartment's voltage level and vice versa. After everything is recorded, communication ends and each compartment performs computations which recalculate their biological parameters, most important of which the voltage level of each compartment's membrane. For each axonal compartment, its new voltage level is recorded in the simulation's output file and the simulation proceeds to the next step. This process repeats until either the input current file ends, which indicates the end of the desired simulation duration, or the hard-coded input spike (120,000 steps) ends, in case no input current file is provided.

4.3 Porting Options

There were two main methods used to distribute the application's workload over the 48 cores of the SCC. The simpler porting option #1 divides the cell network to smaller fragments, each with the same number of cells, and distributes each to a different SCC core. Cells that need to communicate and are mapped to the same core can do so by simply accessing the core's private memory. However, if communicating cells belong to different cores, then these cores need to use the RCCE library [22] to communicate with each other. This allows the sharing of their dendritic voltage levels. This method constitutes strict data partitioning and has the benefit of needing less effort for communication purposes.

For porting option #2, the cores are divided in groups of three, similar to the JPEG Protocol application described in Chapter 3. A different segment of the network is assigned to each group, with each core of the group handling different compartment: dendritic, somatic and axonal duties are mapped to different cores. For this option, each core has different amount and nature of work to complete. The cores handling dendrites tackle most of the communication necessities of the application and reading the input current file. The somatic cores perform heavy computations and communicate with the other two cores in their group. The cores mapped to axonal compartments mostly perform I/O operations. Thus, this porting option is more complicated and features task partitioning on top of data partitioning. However, as it will be shown, it allows for simpler and more efficient techniques to reduce the simulator's power consumption. This was encouraged by the fact that, as shown in Figure 4.2, according to profiling carried out prior to designing porting option #2, approximately 50% of the total computational workload was attributed to the somatic compartment. As such, cores handling somatic duties needed to operate at full capacity, whereas cores with lesser workload could simply be scaled down frequency-wise.

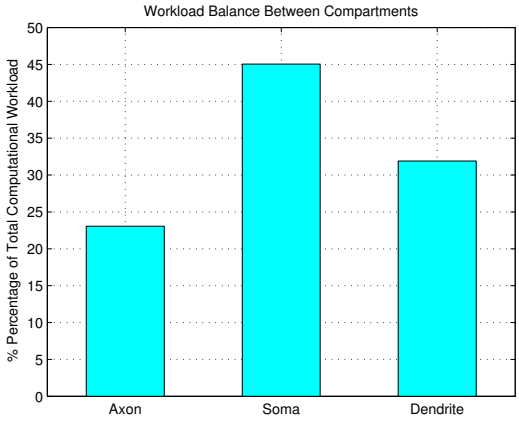

Figure 4.2: Results of IO simulator profiling, detailing workload distribution between cell compartments

Thus, in the interest of pursuing ways of lowering power consumption for long, demanding simulations, for each porting option a method of using the application at reduced power is developed. The starting point in both strategies is the same: as mentioned before, the axonal compartment features significantly lower workload than the other two compartments. Since the dendritic compartments handle most of the communication between cells and the somatic compartments tackle the greatest part of a cell's calculations for each step, the axonal compartment's duties are lighter than the rest. This is particularly true for simulations of bigger networks, where the dendritic compartments take up an increasing degree of computing resources. Thus, using the voltage and frequency scaling utilities of the SCC (`RCCE_iset_power()` [18]), less energy is expended when handling axonal duties.

This is accomplished in different ways for the two options. Porting option #1 is based on each core handling the entirety of a cell's workload. For this option, frequency of operation is dynamically lowered in each simulation step during the computation of the axonal compartment and restored after its end. Voltage levels are not altered since voltage modulating tools are not fast enough to keep up with dynamic manipulation, as opposed to altering the cores' frequency, which happens in a matter of a few clock cycles. Porting option #2 takes a more static approach. Since the axonal compartments are mapped to a particular, predefined group of cores, these cores lower both voltage and frequency of operation before the actual simulation commences. There are necessary communicational barriers in each step so that all compartments are synchronized. Thus, reducing the frequency of low-workload cores does not gravely impact the application's overall performance since the application's bottleneck lies in the cores handling dendritic and somatic compartments. Overall board power consumption and peak power levels are expected to be lower without hampering the application's speed.

4.4 Porting Assessment

Each benchmark simulation has been carried out without an input current file specified. Thus, all simulations use the hard-coded input spike and last for exactly 120,000 steps, or 6 seconds of real brain activity. All parameters have been initialized with the same standard values and random closed-circuit initialization steps have been disabled. For simulations using porting option #1, since it is interesting to see the benefits of using the board to its full capacity of 48 cores, cell grids simulated feature cell numbers which are multiples of 48. For porting option #2, since employing 48 cores allows the application to use 16 groups of three cores, simulated cell grids have total cell numbers which are multiples of 16. It is important to note that all benchmarks have been carried out under the "naive" assumption of 8-way connectivity, thus communication overhead is reduced. The application behaves very differently in case of densely intercommunicating networks. However, inspecting realistic connectivity schemes falls beyond the scope of the current thesis.

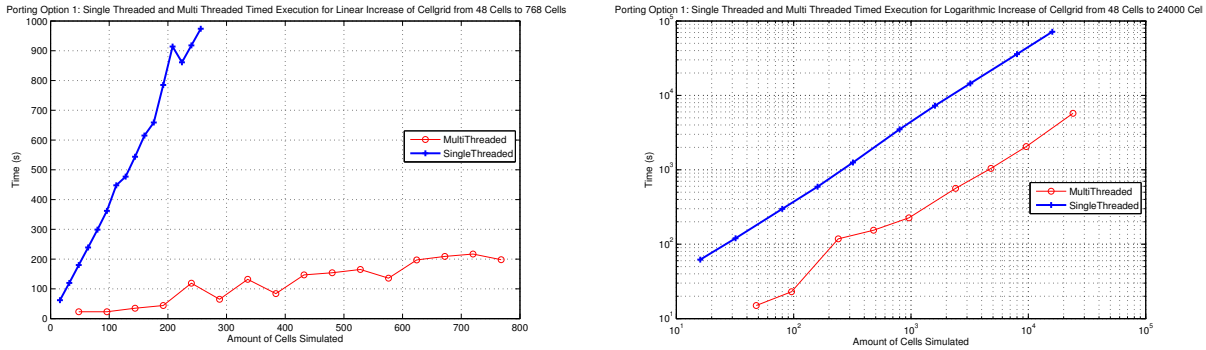

Figure 4.3: Porting option #1: Brief linear and extended logarithmic sweep

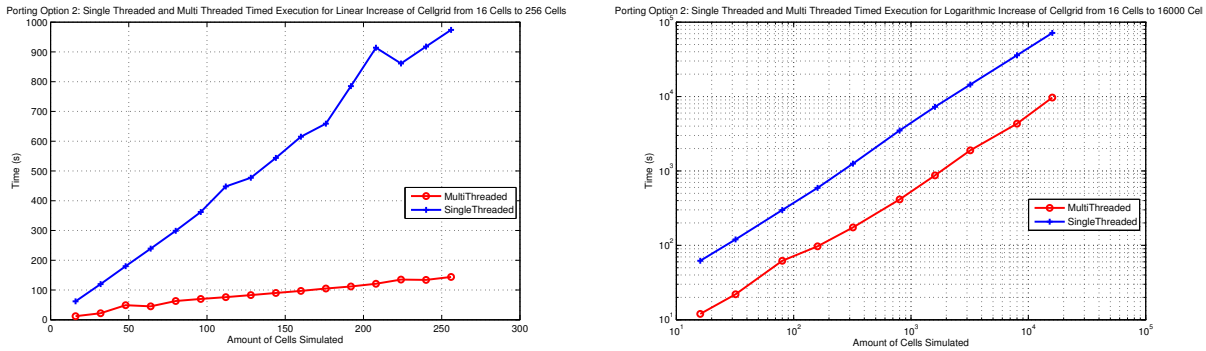

Figure 4.4: Porting option #2: Brief linear and extended logarithmic sweep comparison against single-threaded version

Figures 4.3 and 4.4 demonstrate the two porting options' performance. Figure 4.3 describes how porting option #1 behaves with increasing network grids, both in a linear and a logarithmic

scale. Networks of up to 24,000 cells have been successfully simulated for 6 seconds of brain activity in under 90 minutes. Further increase of cell grid size causes the application to crash due to insufficient memory. Larger network simulations return *rcce error code 137*, which according to Intel documentation [17] signifies a segmentation fault. However it should be noted that up until that point, the application scales in a linear fashion as network size increases. Figure 4.4 showcases the speedup of the application's porting option #2 compared to the single threaded application. Speedups of up to a factor of 8 are achieved for big networks. The Figures also point to the fact that porting option #1 is more than two times "faster" than the second for the same network sizes. Thus, the application, when ported on the SCC, can perform more than 16 times faster than the single-threaded version, assuming one uses all 48 cores of the board and porting option #1. It should also be noted that porting option #2 uses more memory than the first. As a result it crashes for networks significantly larger than 16,000 cells. Another interesting result of these Figures is the fact that porting option #1, while "faster", shows a stochastic performance for smaller cell grids while the second option's results are more linear and predictable. This issue of performance variability [21] can be very important for researchers aiming at simulating grids with less than 1,000 cells.

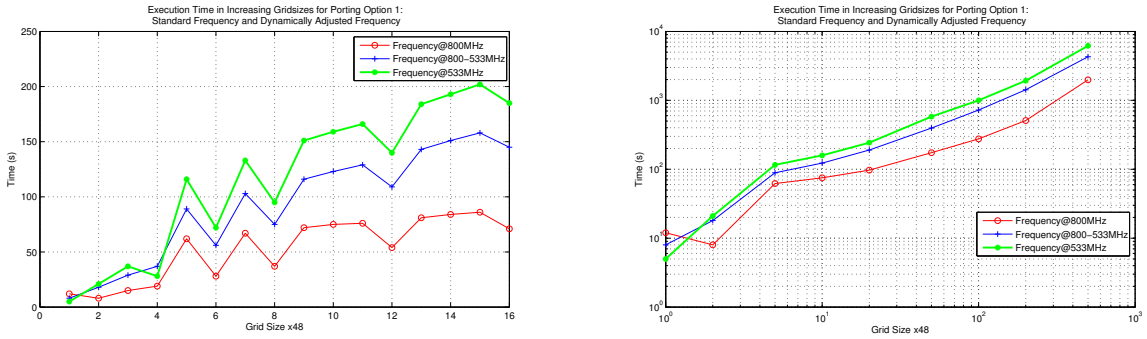

Figure 4.5: Porting option #1: Brief linear and extended logarithmic execution time comparison between standard and DFS mode

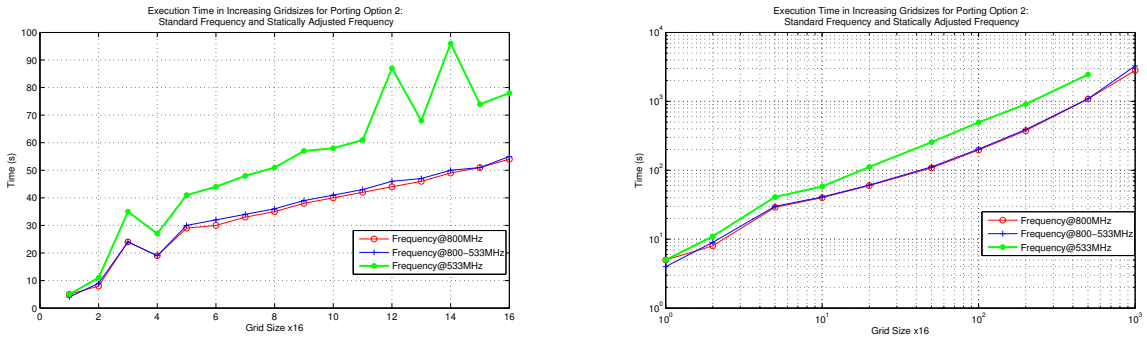

Figure 4.6: Porting option #2: Brief linear and extended logarithmic execution time comparison between standard and SVFS mode

Figures 4.5 and 4.6 detail what impact on execution times do power management methods employed hold. Figure 4.5 describes the performance of porting option #1 when all cores perform at a standard 800MHz, as well as a sub-optimal 533MHz frequency against dynamic frequency scaling (DFS) where all cores operate at 533MHz when performing axonal duties and 800MHz at any other time. It should be noted that for these benchmarks, aiming at comparing performance when applying power management methods, all output has been disabled and as a result, overall performance has improved greatly. This fact also points at the toll heavy I/O operations can take on applications, especially when many cores flood multiple large files with data and when cloud applications run on several different computation machines [28]. The Figures show that DFS decreases the simulator's performance considerably, possibly due to the high number of frequency alterations since it happens twice per simulation step.

This is not the case with Figure 4.6 which describes Static Voltage-Frequency Scaling (SVFS), applicable only to porting option #2, where no notable delay is observed even for large cell networks. Time-wise, static scaling of power levels is superior, for this application, to dynamic scaling and also simpler to employ since no inline changes to the code are necessary and all changes happen before the application commences. Good results by SVFS also reinforce earlier benchmarking, which indicates axonal compartments hold a smaller percentage of overall workload, thus enabling cores on strictly axonal duties to perform adequately even at lower frequencies. It should also be noted that in both cases, simply lowering frequency of operation to 533MHz is not a feasible solution, as performance is gravely impacted, especially in the case of porting option #2.

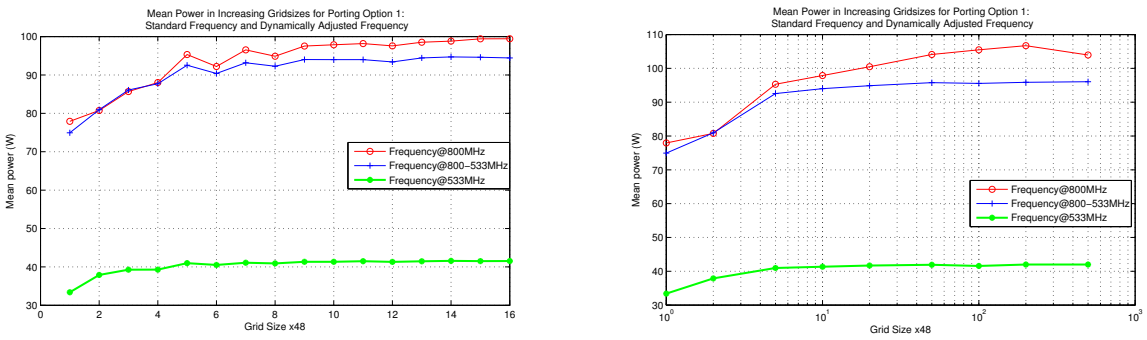

Figure 4.7: Porting option #1: Mean power levels comparison between standard and DFS mode, brief linear and extended logarithmic sweep

Figures 4.7 and 4.8 describe the average power consumption for simulations of increasing grid size. The average levels of power used by the board to function is an important parameter since high levels of power for prolonged periods of time can strain a system and cause it to malfunction or underperform in the long run (for example, mobile terminal batteries deteriorate faster based on cycles of 100% charge and discharge [43]). Striving for lower power levels of operation is to a developer's interest. Both methods successfully lower the average power level

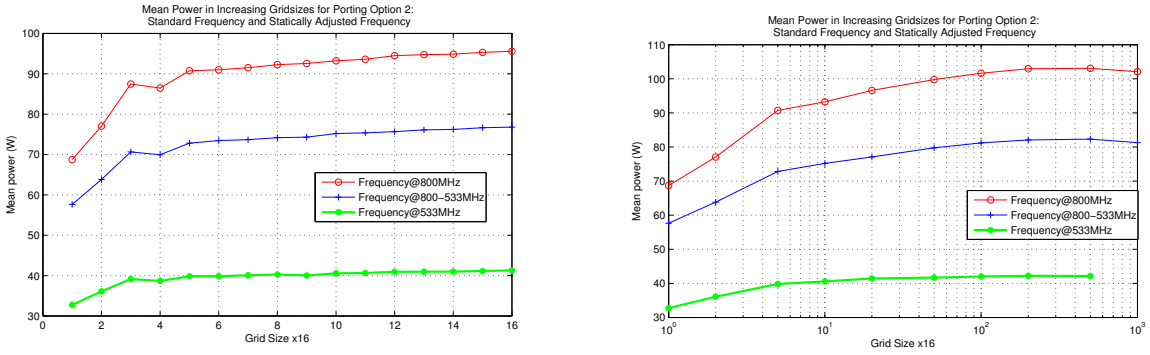

Figure 4.8: Porting option #2: Mean power levels comparison between standard and SVFS mode, brief linear and extended logarithmic sweep

maintained throughout the simulations. However, one observes that SVFS on porting option #2 is more effective than DFS on the first option. SVFS reduces power levels by a more noticeable amount and does so more reliantly, whereas DFS exhibits unreliable performance for small network simulations. It should also be noted that the highest average power for SVFS of porting option #2 is much lower than in DFS case of porting option #1 (80-85 Watts against 95-100 Watts). While operating at 533MHz yields considerable benefits, the unavoidable impact on performance, as observed in Figures 4.5 and 4.6 render this option unusable for most potential users of the simulator.

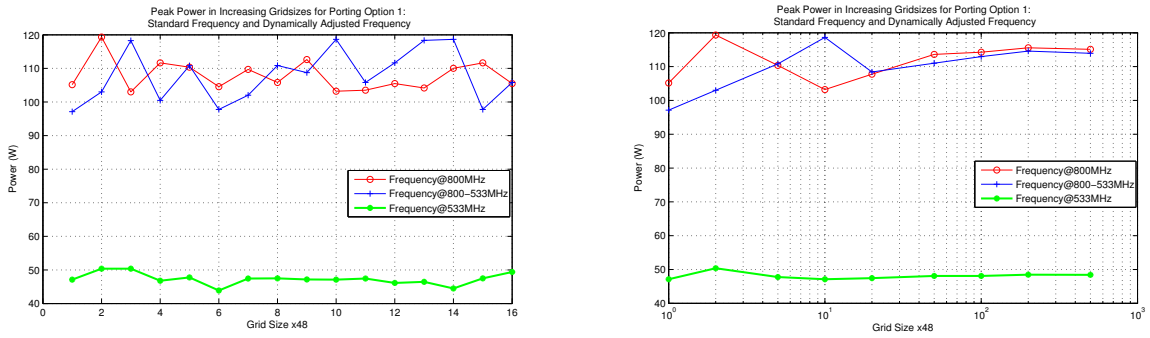

Figure 4.9: Porting option #1: Peak power levels comparison between standard and DFS mode, brief linear and extended logarithmic sweep

Figures 4.9 and 4.10 show another important aspect of power management, peak power. It is well documented how sudden fluctuations of power levels can damage equipment. While hardware solutions exist (such as voltage regulators [13]), software tools that allow power management can prove useful in designing applications that attempt to reduce such stress by guaranteeing low upper thresholds in power usage. Such software tools are part of a wider attempt to lower energy requirements and consumption for applications globally [45]. Much like previous Figures 4.7 and 4.8, it is shown that SVFS outperforms DFS. DFS actually presents power spikes worse than the standard case of porting option #1 for small networks and highly

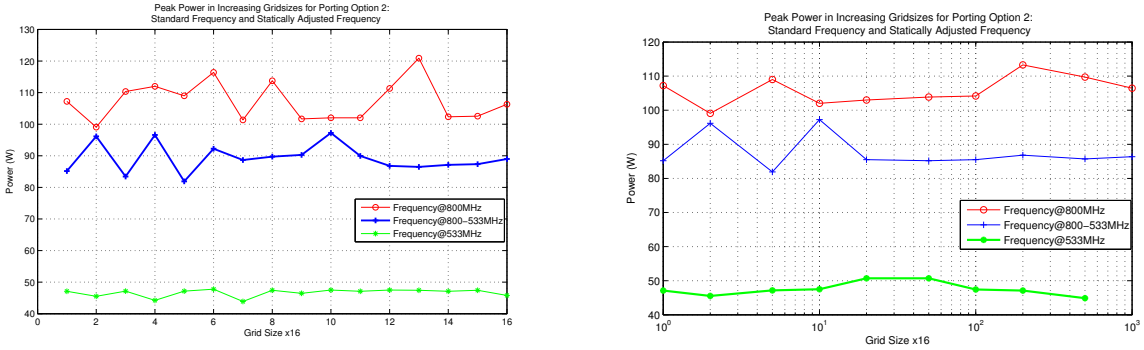

Figure 4.10: Porting option #2: Peak power levels comparison between standard and SVFS mode, brief linear and extended logarithmic sweep

unpredictable behaviour. It is deemed unsuitable for the purposes of controlling peak power levels. Figure 4.10 shows that SVFS succeeds in reducing maximum power levels, however it also presents a less than ideal behaviour concerning reliably and predictably lowering power spikes as cell networks grow larger.

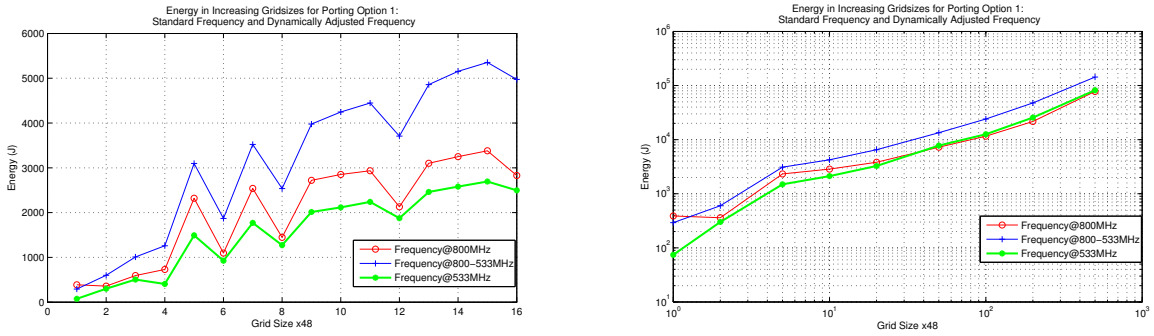

Figure 4.11: Porting option #1: Energy expenditure comparison between standard and DFS mode, brief linear and extended logarithmic sweep

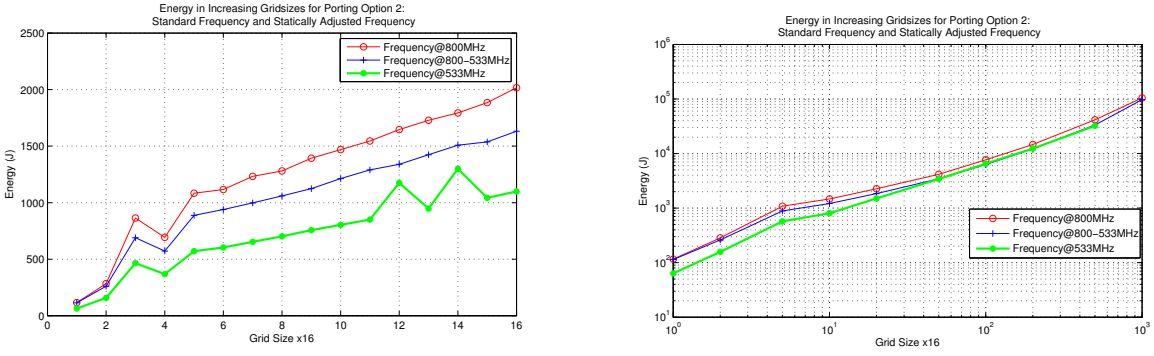

Figure 4.12: Porting option #2: Energy expenditure comparison between standard and SVFS mode, brief linear and extended logarithmic sweep

Figures 4.11 and 4.12 detail actual energy consumption throughout the completion of

increasing in size simulations. Energy consumption has been measured by integrating power levels over the duration of each simulation. These Figures show that DFS for porting option #1 is overall ineffective. The fact that previous Figures indicate DFS hampering the application's performance, while not providing reliable power level benefits, supports the results of Figure 4.11. Energy expenditure is actually worse in most cases, proving that this method is inefficient for this application. However, Figure 4.12 shows promising results for SVFS of porting option #2. Power expenditure is always lower when SVFS methods are applied and for relatively small networks (less than 250 cells) the benefits are noteworthy, even up to 25 % reduced energy consumption. In combination with reduced average power and maximum power spike levels, while not noticeably affecting the application's speed, SVFS can be described as an effective and simple method of power management, provided the application's architecture supports it. Finally, these Figures prove that simply operating at 533MHz is not an efficient solution for energy savings, since it is not directly associated with lower energy expenditure (especially for large network simulations). This can be attributed to the fact that, while power consumption is lowered, each simulation runs for much longer due to second-rate performance, thus nullifying any reasonable energy gain.

Chapter 5

General Conclusions and Remarks

The current thesis aims at using the SCC as a concept vehicle for the porting of two applications, a JPEG Protocol Encoder and an Inferior Olive Simulator. The main goals were to achieve massive speedups compared to respective single-threaded versions by efficiently utilizing the chip's 48 cores, as well as exploring methods of reducing energy expenditure without hampering performance. Both applications yielded interesting results, with the IO Simulator being the main focus of the thesis.

The JPEG encoder project proved that HPC can help achieve compression rates that satisfy today's criteria of acceptable FPS. Utilizing more cores leads to better performance at the cost of power consumption. However, this is a general conclusion one is lead to when using HPC on many-core platforms; the developer enjoys the freedom of suiting the parameters of an application's execution to his needs, balancing time perfomance and energy costs. This balance can be accomplished in a number of ways, from using a specific portion of a platform's cores to altering voltage and frequency of operation. These methods supply the user with many interesting options, such as keeping cores at idle mode to conserve power but also have cores act as "back-up", in case part of the platform's cores goes offline or becomes damaged (due to power outages or temporal fluctuations in power supply for example).

The user can also use task partitioning more effectively with increasing number of cores at his disposal. Not all tasks are of the same "weight" in a typical application, nor of the same nature as some may be memory-bound and others may be heavily affected by CPU computational performance. Thus, there is an incentive for building custom multi-core platforms specialized for demanding applications. Cores would be organized in groups handling different tasks optimally, each at different power levels of operation, for maximum performance and power efficiency.

The IO application showcases two different efforts to solve the same problem: strict data partitioning against a task-data partitioning approach. The simpler porting option #1, which only features data partitioning, proves to be faster, particularly when simulating large networks, by a factor of at least 2. If speed is the only concern of the user, then it is the prefered method.

However, DFS methods used in this thesis for porting option #1 did not yield acceptable results. Statically lowering frequency on all cores for both porting options hinders the simulator's performance greatly and does not ultimately offer considerable energy gains, thus it is not a feasible option either. Porting option #2 on the other hand, offers an effective and simple way of reducing power levels considerably. While keeping performance relatively intact, SVFS manages to lower average and peak power levels, as well as offering intriguing energy gains for medium sized networks. As with the JPEG encoder project, it is also more customizable, as a system tailored to the demands of the application can be developed. Such system would have three groups of cores:

- **Group 1** would consist of cores equipped with optimized MPBs for rapid communication purposes. The cores should be physically adjacent to each other to reduce message passing delays even further. They would also require more buffer memory in case of densely intercommunicating networks.

- **Group 2** would need cores featuring the best ALUs available, since they handle the biggest part of the network's computations. Since these cores communicate with both group 1 and 3, it would be suggested that they are physically placed between these two core groups.

- **Group 3** would finally have cores of lower requirements, operating at lower frequency and voltage settings. These cores would benefit greatly from optimized I/O operations, in case axonal voltage levels remains the application's main point of interest.

Such a system would perform remarkably well while providing energy gains, but is only possible to develop while using the more complex porting option #2. As future work, it is very interesting to see how the simulator's behaviour and needs change depending on how densely the network's cells communicate with each other. Not only would the simulator be closer to real brain cell networks, it is very possible that the increased communication overhead shifts the application's bottleneck from the somatic to the dendritic compartment. If that were the case, new porting scenarios and power management options could be considered.

Overall, the SCC has been an experiment at HPC, a platform of one single chip containing many fully operational and autonomous processing centers, like a Cloud. Since its original purpose was exploring the limits of HPC and teaching researchers how to create code for multi-core systems, its programming paradigm was very simple and easy to learn. It is a platform allowing the developer to use methods described above for power management. Software-controlled voltage and frequency of operation on different cores is an invaluable and effective tool. It certainly points to a future where power efficiency is of paramount importance for "heavy" applications. As a general remark, the author believes that shifting this control to the programmer's side is a step towards the right direction.

Bibliography

[1] http://bluebrain.epfl.ch/. 19

[2] http://communities.intel.com/community/marc. 8, 9

[3] http://documentation.apple.com/en/cinematools/usermanual/#chapter=2%26section=5. 15

[4] http://en.community.dell.com/techcenter/high-performance-computing/w/wiki/using-high-performance-computing-for-personalized-medicine.aspx. 12

[5] http://infai.org/jpeg?show_comments=1. 10

[6] http://www.hpcfinance.eu/projects. 12

[7] http://www.ietf.org/rfc/rfc4251.txt. 8

[8] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html. 9

[9] http://www.jpeg.org/jpeg/index.html. 10

[10] http://www.open-mpi.org/. 6, 9

[11] http://www.rfc-editor.org/rfc/rfc2435.txt. 18

[12] http://www.videotechnology.com/0904/formats.html. 15

[13] *Linear & Switching Voltage Regulator Handbook*. 28

[14] www.es.ele.tue.nl/mininoc/. 12

[15] The scc programmer's guide, revision 0.75. Technical report, Intel Labs, 2010, October 19. 4, 6

[16] The scc platform overview, revision 0.75. Technical report, Intel Labs, 2010, September 1. iii, 3, 4, 5, 8

[17] Error codes. Technical report, Intel Labs, 2011, May 9. 7, 26

[18] Using the rcce power management calls, revision 1.1. Technical report, Intel Labs, 2011, September 27. 8, 24

[19] *High performance computing for parallel rendering in surgical stereoscopic display and navigation*, June 2003. 12

[20] Behind the scenes. *Intel premier IT Magazine*, winter 2008. 12

[21] Thomas E. Bell. Computer performance variability. In *AFIPS '74 Proceedings of the May 6-10, 1974, national computer conference and exposition.* 16, 26

[22] Ernie Chan. Rcce_comm: A collective communication library for the intel single-chip cloud computer. Technical report, 2010. 6, 18, 23

[23] Boris Kovatchev Ph.D. et al. Control to range diabetes: Functionality and modular architecture. *Journal of Diabetes Science and Technology*, 2009, September. 19

[24] Carsten Clauss et al. ircce: A non-blocking communication extension to the rcce communication library for the intel single-chip cloud computer. Technical report, RWTH Aachen University, 2011, November 4. 6, 8

[25] Chris I. De Zeeuw et al. Microcircuitry and function of the inferior olive. *Trends In Neuroscience*, 1998. 19

[26] F. Benedetti et al. Inferior olive lesion induces long-lasting functional modification in the purkinje cells. *Experimental Brain Research*, 1984. 19

[27] Howard J. et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *Solid-State Circuits, IEEE Journal of*, 2011. 1

[28] Lofstead J. et al. Adaptable metadata rich io methods for portable high performance io. *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2012. 27

[29] Radulescu A. et al. Cpr: Mixed task and data parallel scheduling for distributed systems. In *Parallel and Distributed Processing Symposium., Proceedings 15th International.* 1

[30] Timothy G. Mattson et al. The 48 core scc processor: A programmer's view. Technical report, Intel Corporation. iii, 3, 4

[31] Rick C. Hodgin. Background: Intel's tera-scale project. *TGDaily*, 2007, September 1. 3

[32] Sebastian Isaza. Olivocerebellar hardware modelling in the fpga lab. Technical report, Erasmus MC - Dept. of Neuroscience, 2012, June 18. 20

[33] Ringo J.L. Neuronal interconnection as a function of brain size. *Brain, Behavior and Evolution*, 1991. 21

[34] Intel Labs. The intel xeon phi product family: Highly parallel processing for unparalleled discovery. 9

[35] Intel Labs. Using the sensor registers. Technical report, 2012, December 21. 5

[36] Didier Fuin Luca Benini, Eric Flamand and Diego Melpignano. P2012: building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceeding DATE '12 Proceedings of the Conference on Design, Automation and Test in Europe*. 9

[37] Randall C. O'Reilly and Yuko Munakata. *Computational Explorations in Cognitive Neuroscience: Understanding the Mind by Simulating the Brain*. 2000. 1, 19

[38] Vilfredo Pareto. *Manual of Political Economy*. Augustus M. Kelley Publishers, 1971. 17

[39] Li Ou PhD. Parallel rendering technologies for hpc clusters. *DELL Power Solutions*, November 2007. 12

[40] Stefan M. Wild Prasanna Balaprakash, Ananta Tiwari. Multi-objective optimization of hpc kernels for performance, power and energy, 2013, April. 8

[41] D. Rodopoulos, A. Papanikolaou, F. Catthoor, and D. Soudris. Software mitigation of transient errors on the single-chip cloud computer. In *IEEE Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2012. 13

[42] Erich Strohmaier Shoaib Kamil, John Shalf. Power efficiency in high performance computing. 20

[43] Kazuhiko Takeno and Remi Shirota. Capacity deterioration characteristics of li-ion batteries for mobile terminals. *NTT DoCoMo Technical Journal*. 27

[44] Tektronix. A guide to mpeg fundamentals and protocol analysis. 18

[45] Rajesh K. Gupta Vijay Raghunathan, Mani B. Srivastava. A survey of techniques for energy efficient on-chip communication. In *DAC '03 Proceedings of the 40th annual Design Automation Conference*. 28

[46] Gregory K. Wallace. The jpeg still picture compression standard. Technical report, Digital Image and Video Standards. 10

[47] Susan Young. Brain implant detects, responds to epilepsy. *MIT Technology Review*, 2012, October 12. 19