Development, Implementation, Optimization and Performance Analysis of Matrix-Vector Multiplication on Eight-Core Digital Signal Processor

FERUZ MURADOV

Master of Science Thesis
Stockholm, Sweden 2013
Development, Implementation, Optimization and Performance Analysis of Matrix-Vector Multiplication on Eight-Core Digital Signal Processor

FERUZ MURADO

Master's Thesis in Scientific Computing (30 ECTS credits)
Master Programme in Scientific Computing (120 credits)
Royal Institute of Technology year 2013
Supervisor at KTH was Stefano Markidis
Examiner was Michael Hanke

TRITA-MAT-E 2013:50
ISRN-KTH/MAT/E--13/50--SE

Royal Institute of Technology
School of Engineering Sciences
KTH SCI
SE-100 44 Stockholm, Sweden
URL: www.kth.se/sci
Abstract

This thesis work aims at implementing the sparse matrix vector multiplication on eight-core Digital Signal Processor (DSP) and giving insights on how to optimize matrix multiplication on DSP to achieve high energy efficiency. We used two sparse matrix formats: the Compressed Sparse Row (CSR) and the Block Compressed Sparse Row (BCSR) formats. We carried out loop unrolling optimization of the naive algorithm. In addition, we implemented the Register-blocked and the Cache-blocked sparse matrix vector multiplications to optimize the naive algorithm.

The computational performance improvement with loop unrolling technique was promising (≈ 12%). With this optimization, we observed a decrease of power usage (0.3 W) when using a matrix size of 600 and an increase of power usage (1.2 W), when using larger size matrices. The Register-blocked algorithm resulted to be the most efficient technique on DSP. With this algorithm, we were able to increase performance by a factor of six when compared to the naive algorithm, still retaining low power consumption (≈ 14 W). The Cache-blocked sparse matrix vector multiplication is known to be most convenient for large number of architectures with coherent caches. However, because DSP does not support coherency between caches, this method did not show large improvement in computational performance. In fact, we confirm that power consumption for the Cache-blocked method was higher when compared to other effective algorithms such as Register-blocked sparse matrix vector multiplication and loop unrolling of naive algorithm. In conclusion, we found that the DSP delivers low power consumption, excellent computational performance and energy efficiency when the Register-blocked sparse matrix vector multiplication technique is used.
Referat

**Utveckling, genomförande, optimering och Prestandanlys av Matrisvektormultiplikation på åtta-kärnors Digital Signal Processor**


To my dear family and
my mentor, Sulaymon Eshkabilov
# Contents

1 Introduction 1  
   1.1 Motivation .......................................................... 1  
   1.2 Goals .................................................................. 3  

2 Literature Survey 5  

3 Matrix vector multiplication algorithms 7  
   3.1 Different sparse matrix formats .................................. 7  
      3.1.1 Compressed Sparse Row format ......................... 8  
      3.1.2 Block Compressed Sparse Row format ................. 8  
      3.1.3 Cache-blocked storage format .......................... 9  
   3.2 Sparse matrix vector multiplication algorithms ........... 9  
   3.3 Architecture of Digital Signal Processors ................. 10  
   3.4 Programming environment on Digital Signal Processor ... 13  
   3.5 Sparsity, flops, and energy efficiency measures ......... 15  

4 Implementation of sparse matrix vector multiplication 17  
   4.1 Naive sparse matrix vector multiplication ................. 17  
   4.2 Register-blocked sparse matrix vector multiplication .... 18  
   4.3 Cache-blocked sparse matrix vector multiplication ....... 18  
   4.4 Cache coherence on Digital Signal Processors .......... 19  
   4.5 Parallelization of sparse matrix vector multiplication ... 21  

5 Optimizations for sparse matrix vector multiplication 23  
   5.1 Optimize Algorithms for Digital Signal Processor .......... 23  
   5.2 Parallel optimization ............................................. 29  

6 Measurements and Results 31  
   6.1 Post-processing scripts ......................................... 31  
   6.2 Energy and Performance results ............................... 34  

7 Discussion and Conclusions 43  
   7.1 Serial Implementation ............................................ 43  
   7.2 Parallel Implementation ........................................... 43
Acknowledgements

I would like to express my gratitude to my teacher and supervisor, Dr. Stefano Markidis, for his continuous guidance, patience, and encouragement that have been invaluable on both academic and personal levels. His insightful advice and unsurpassed knowledge kept me focused on my goals.

I wish to express my sincere thanks to all my teachers at NADA and PDC, specially Dr. Lennart Edsberg for his help and encouragement throughout my study, Dr. Michael Hanke, Prof. Jesper Oppelstrup for their deep knowledge and Ms. Beatrice Frock, Ms. Carina Edlund for friendliness that made the environment at the classroom both constructive and fun. Their insightful experience and rich knowledge of scientific computing, multi-core programming provided me with enthusiasm and comprehension.

It is with immense gratitude that I acknowledge the support and help of grant holder Mr. Lennart Johansson and coordinator Dr. Sulaymon Eshkabilov. Financial support was provided by CD_JEP 26123-2005 project. My deep gratitude to IT Center staff for their help and encouragement. Their fruitful discussions, and energy have been a tremendous source of inspiration.

I acknowledge the help and support given to me by Prof. Lennart Johnsson at PDC/KTH and deeply thank Gilber Netzer, PhD, student at PDC and the provider of the tools, for his support, background and contributions.

Finally, I owe my deepest gratitude to my family and friends for their love and support at all times. I am most grateful to my parents, sister and brother for helping me to be where I am now.
Chapter 1

Introduction

Digital Signal Processors (DSP) are specialized microprocessors with an architecture optimized for low power consumption. DSPs are used for audio/video processing, portable and mobile devices as well as many embedded systems. This type of microprocessors has a different approach for hardware design, and requires irregular software optimization to achieve good scaling performance. Therefore, it is essential for software developer to optimize algorithms by hand in assembly code routines. In modern days, processors require no extra effort to optimize code, because advanced and sophisticated compilers handle optimization for them. But modern processors require to use more chips or microschemas which consume extra power.

The most attractive feature of DSP is the low energy consumption. DSP provides lower cost solution with better performance and low latency (no support for virtual memory, virtual memory requires more time for context switching between processes which results in increased latency).

The use of DSP in High Performance Computing (HPC) has being proposed by scientific community because of power efficiency and good computational performance.

In this thesis, we test DSP with Sparse Matrix-Vector multiplication (SpMV) algorithms to measure power consumption and performance.

The thesis is organized as follows. In Chapter 1, we provide an introduction to this work including the motivation and goals of this thesis. In Chapter 2, we describe previous work in the field. In Chapter 3, we presents different matrix vector multiplication algorithms. In the Chapter 4, we provide implementations of algorithms. Optimizations for SPMV are described in Chapter 5. The measurements and results of the experiments are presented in Chapter 6. Finally, we discuss and summarize the results in Chapter 7.

1.1 Motivation

The main motivation for this thesis project is to study and investigate the performance of matrix vector multiplication on a Texas Instrument’s (TI) DSP device. In particular, we focus on studying its energy efficiency. In fact, lower power usage is
one of the fundamental requirements of our daily computers, smart-phones, tablets and many more devices. Power usage for modern daily personal computers can be in a range of 300 W to more than 1000 W for high performance computers with multiple graphics and Central Processor Unit (CPU) cores. General purpose computers require approximately 300-500 W \[1\]. As the demand on high performance systems increases, power consumption increases as well. This occurs because the power consumption depends on the clock frequency. For a given device, operating at a higher clock rate always requires more power. In most architectures it is possible to reduce the clock rate of the microprocessor through power management leading to reduced power consumption. But new features generally require more transistors, each of which uses power as well. As a processor model’s design develops, smaller transistors, lower-voltage structures, and design experience may reduce energy consumption. In terms of units, processor manufacturers usually release two power consumption numbers for a CPU: typical thermal power, which is measured under normal load (for instance, Advanced Micro Devices AMD’s Average CPU power (ACP)) and maximum thermal power, which is measured under a worst-case load. For example, the Pentium 4, 1.6 GHz has 24.5 W typical thermal power. A more recent architecture, the QuadCore Processor with GT2 graphics has minimum and a maximum Thermal Design Power \[2\] of 37 W and 47 W respectively. When the CPU idles, it will draw far less than the typical thermal power.

According to Ohm’s law, the power consumed by a CPU is approximately proportional to CPU frequency, and to the square of the CPU voltage \[3\]:

\[
P = CV^2f
\]

(where C is the capacitance, f is the frequency and V is the voltage). The DSP device can be modeled as a series of connected capacitors. The amount of energy in each capacitor is \(\frac{1}{2}C_i \cdot V^2\). If each capacitor is charged and discharged \(f\) times per second, the exchanged energy is \(\frac{1}{2}C_i \cdot V^2 \cdot f\). Summing for all switching capacitors and substituting \(C = \sum \frac{C_i}{2}\), we obtain \(C \cdot V^2 \cdot f\).

One of the goals of TI is to minimize the power consumption and to reach high computational performance. Manufacture achieves these goals by designing very complex digital signal processors (DSP). It includes both chip design and manufacturing processes. As an example, Power consumption on TI DSP C6678 for double-precision 60 Flops requires 10 watts on chip, 50 watts on board, which can also give 160 Flops for single-precision \[4\]. Some TI literature uses 10 W for just core \[5\], while the packaging specification shows 24 W absolute max, we can assume that \(\pm 7\) W from overall 24 W is the Thermal Design Power (TDP) of the TI chip.

Table 1.1 presents the power consumption and relative clock frequency in some common processors.

\(1\) TDP measure usually refer to maximum amount of Power used for cooling system of computers or processors
Numerical methods are computational techniques used in scientific computing, signal and image processing and many other applications. Many real-world problems are based on matrix operations (e.g. Matrix matrix multiplication and Matrix vector multiplication). This operation is often used in explicit methods for ordinary and partial differential equations, in eigenvalue problems as well as solving linear systems of equations with iterative methods. Sparse matrix operations are common in scientific methods especially with iterative methods to solve linear system of equations. Currently, there is no convenient way or benchmarking to measure the performance of the TI DSPs. This thesis project provides initial performance and power efficiency results of the TI DSPs.

1.2 Goals

The goals of this thesis are the following:

- To develop and implement matrix-vector multiplication algorithms in a DSP system.
- To study and analyze the performance of matrix-vector multiplication varying the matrix size and the matrix sparsity (number of null entries) on DSP.
- To provide a user-friendly benchmark that can be used in future implementations of algorithms on DSP systems.
- To investigate and analyze the power efficiency of matrix-vector multiplication on a DSP.
- To understand the possible role and convenience of using DSP systems in HPC by evaluating the difficulty of programming DSP and the power efficiency of these devices.
Chapter 2

Literature Survey

In this thesis, we focus on the development, implementation and optimization of sparse matrix-vector algorithm on DSP. DSP devices have been mainly used for audio/video encoding [6], image processing (requires heavy matrix operations) [7], as well as real-time applications which includes video streaming [8], diagnostic medical imaging [9], sensor measurements and test [10], and many more applications [11]. Energy-efficient algorithms have been developed in order to use lower energy in HPC algorithms. For instance, in multithread or multicore processing one can optimize processor usage knowing if the system is active or idling. Algorithms have been developed to manage the processors sleep/active states in order to save power.

Recently, there have been some proposals about benchmarking DSP and achieving higher performance still using low energy. The effective use of computational architectures with accelerators requires a wide system knowledge as well as irregular programming methods and skills. These systems are composed of commodity processors integrated with Field Programmable Gate Arrays (FPGA) and/or Graphics Programming Units (GPU). DSP architecture is one of this type of processors. One advantage of the DSP is that it avoids the performance problems associated with integrating accelerators into computer systems. The TMS320C6678 multi-core DSP promised to be power, cost efficiency and hope to provide same programming paradigm as multi-core CPUs [5].

One of the driving factor of development for DSP processors is that increase speed, decrease energy consumption, as well as decrease memory usage, and decrease cost [12].

Investigations of some researchers from Berkeley University of California indicates that, ... shallower pipelines with in-order execution have proven to be the most area and energy efficient. Given these physical and micro architectural considerations, we believe the efficient building blocks of future architectures are likely to be simple, modestly pipelined (5-9 stages) processors, floating point units, vector, and SIMD processing elements. [13]. This outcome from the investigation shows that the DSP microprocessor from Texas Instruments embodies this recommendation effective among AMD International Business Machines (IBM), Intel, NVIDIA
chips.

As an application, we have implemented and tested Sparse Matrix-Vector multiplication (SpMV) for this thesis. This numerical problem scales poorly on modern multicore architectures because of its high demands of memory bandwidth. One could add more threads and keep the processor frequency high but it might not accomplish the expected performance gains. Leading to a waste of energy.

Some researchers from National Technical University of Athens attempted to optimize Sparse Matrix-Vector Multiplication with simple power metric. They attempt to rank the different configurations to power dissipation and use the theory of multi-objective optimization to analyze and characterize the trade-offs. They propose a simple prediction model which will predict best configurations for performance-energy trade-offs for SpMV execution. [14]

There are a number of different libraries also trying to optimize matrix vector multiplication and matrix matrix multiplication. Among them, the most famous is the Basic Linear Algebra Subprograms (BLAS) standard, where matrix vector operations form the level two BLAS family [15], and matrix-matrix operations form the level three BLAS family [16].

BLAS library has good scaling performance but it targets certain architecture of processors. The main concern of these libraries is the performance for specific architectures. In this thesis project we optimize sparse matrix vector multiplication for eight core TI's DSP. In addition, we investigate optimization techniques for DSP programming, aiming at increasing computational efficiency and lowering energy consumption.
Chapter 3

Matrix vector multiplication algorithms

Sparse matrices have different structures in terms of their non-zero elements or problem types. Therefore it is essential to optimize their structure to use memory and computing units effectively. We start this chapter, presenting the sparse matrix formats and their differences.

Differences in sparse matrix formats require special algorithms because of the change of matrix structure. We describe some of the naive algorithms for each sparse matrix formats. Afterwards, we review the DSP processor architecture, the main advantages and disadvantages of the DSP devices. The technical details and programming environment are described. In addition we present what tools help developers to compile the source and how DSP compiler works.

We finish this chapter with a brief description of the performance parameters used to characterize the DSP efficiency.

3.1 Different sparse matrix formats

Sparse matrices are matrices where the majority of elements is zero. Sparse matrices are usually obtained from discretization of partial differential equations and many other applications. They are widely used in scientific computations and real world problems. In this thesis project, we only describe sparse matrix vector multiplication in particular processor architectures.

While a dense matrix is easy to be represented as either two dimensional or one dimensional array, a sparse matrix has slightly different structure. Since the majority of elements in sparse matrix is zero, only non-zero elements are stored in different storage formats or structure.

Among the most common sparse storage formats in the scientific communi-
ties [17], we describe the Compressed Sparse Row (CSR), Block Compressed Sparse Row (BCSR) and some additional techniques to get high efficiency for a given architecture.
3.1.1 Compressed Sparse Row format

CSR is one of the basic data matrix structure for storing sparse matrices. The main idea behind this structure shown in Figure 3.1. Structure holds only non-zero entries from matrix in row major order. rowp vector indicates where each sparse vector begins in indx vector and indx vector indicates column index of non-zero element in matrix and matvals holds all non-zero elements in matrix.

Figure 3.1. Example of Compressed Sparse Row format

3.1.2 Block Compressed Sparse Row format

BCSR has slightly different structure from CSR. Here, each vector that represents a sparse matrix is differently arranged. We can see a simple example of this structure in Figure 3.2. The rowp vector points to the block row starting position in indx, while indx stores only first column index of each block, matvals vector stores matrix values as $2 \times 2$ dense block fashion. The goal is to increase register usage in every cycle. For this reason it is referred as Register Blocking.

Figure 3.2. Example of Block Compressed Sparse Row format with $2 \times 2$ dense block
3.1.3 Cache-blocked storage format

Another storage format is the Cache-blocked storage format for sparse matrices. Here the sparse structure remains the same but it rearranges computation so that the blocks of values in the matrix are accessed in sequence in time. Figure 3.3 shows the representation of the format.

Figure 3.3. Example of Cache-Blocked sparse matrix storage

3.2 Sparse matrix vector multiplication algorithms

The matrix vector product $y = Ax$ can be represented as:

$$y_i = \sum_j a_{i,j}x_j \quad (3.1)$$

The algorithm traverses the rows of the matrix $A$ with $n \times n$ size. The matrix vector multiplication for the CSR sparse formats can be calculated as:

for $i = 1, n$

\[
\begin{align*}
    y(i) & = 0 \\
    \text{for } j = \text{row_ptr}(i), \text{row_ptr}(i+1) - 1 \\
    y(i) & = y(i) + \text{val}(j) \ast x(\text{col_ind}(j)) \\
    \text{end}; \\
    \text{end;}
\end{align*}
\]

Since this method only multiplies nonzero matrix entries/values, the operation count is two times the number of nonzero elements in $A$, which is a significant savings over the dense operation requirement of $2n^2$.

Similarly, if sparse matrix consists of square dense blocks in some regular pattern, one can modify the CSR format to make full use of such block patterns. This type of sparse matrices typically arises from the discretization of partial differential equations in which there are several degrees of freedom associated with a grid point and abbreviated as Block Compressed Sparse Row (BCSR).
This technique partitions the matrix in small blocks with a size equal to the number of degrees of freedom, and treats each block as a dense matrix, even though it may have some zeros. If $nnzb$ number of nonzero blocks in $n \times n$ matrix, and $b_n$ is the dimension of the each block. Total needed storage is $nnz = nnzb \cdot b_n^2$. The block dimension $d_n$ of $A$ matrix is then defined by $d_n = n/b_n$. The nonzero blocks stored in row-wise fashion and savings in storage locations and reduction in indirect addressing for BCSR over CSR can be significant for the matrices with large $b_n$:

```
for i = 1, nnzb
    for j = row_ptr(i), row_ptr(i+1) - 1
        for ii=1, b_n
            for jj=1, b_n
                y(ii) = y(ii) + val(ii*b_n+jj) * x ((*col_ind)+jj);
            end;
        end;
    end;
end;
```

By finding optimal block size $b_n$ for specific matrices, this algorithm shows that it uses resources effectively and can achieve good performance for certain architectures. In later chapters about optimization, we will discuss in details about advantages and disadvantages of the algorithms.

### 3.3 Architecture of Digital Signal Processors

Our computational experiments run on the following processors: TMS320C6670 and TMS320C6678. The TMS320C6670 Multicore Fixed and Floating Point System on a Chip is a member of the C66x SoC family based on TI’s new KeyStone Multicore SoC Architecture designed specifically for high performance applications. C6670 integrated with four C66x CorePac [DSP], each core runs at 1.0 to 1.20 GHz enabling up to 4.8 GHz.

The TMS320C6678 Multicore Fixed and Floating Point Digital Signal Processor from the same C66x SoC family with eight cores where each core runs at 1.0 to 1.5 GHz enabling up to 10 GHz. The device supports high-performance signal processing applications.

The C66x CorePac [DSP] is fully backward compatible with all existing C6000 family of fixed and floating point [DSP],. There are no major differences between two architectures but we focus on TMS320C6678 platform architecture, see Figure 3.3. The family of C66x microprocessor is an advanced Very Long Instruction Word (VLIW) architecture with eight functional units [19]. There are two multipliers unit and six arithmetic units that can operate in parallel (pipeline can dispatch eight parallel instructions every cycle), implementing 64 or in general purpose 32-bit registers and eight functional units. Architecture supports single-precision and double-precision instructions. There are two general-purpose register files (A and B),
eight functional units (.L1, .L2, .S1, .S2, .M1, .M2, .D1, and .D2), four functional units in each register file.

The .M1 functional units perform all multiply operations. The .S and .L units perform a general set of arithmetic, logical, and branch functions, these .L and .S units can be used for conversions between from/to integer to/from single-precision floating operations as well. The .D units primarily load data from memory to the register file and store results from the register file into memory.

The processor has L1 program, data memories as a cache/sram and shared, local L2 cache/sram as well as external memory interfaces. Figure 3.4 shows Functional Block Diagram of TMS320C6678 device [18]. One can see that given memories L1D cache - 32 KB, L1P cache - 32 KB and L2 cache - 512 KB as well as Multicore Shared Memory (MSM) - 4 MB large enough to solve relatively large problems. Beside each core can access other core’s L2 cache, not L1, which require developer to handle cache coherency.

The Multicore Navigator has been designed to support hardware-assisted functional acceleration that utilizes an innovative packet-based hardware subsystem in KeyStone devices. With an extensive series of more than 8,000 queues and a packet-
aware Direct Memory Access (DMA) controller, it optimizes the packet-based communications of the on-chip cores by practically eliminating all copy operations. This gives extreme efficiency to gain 100X performance in terms of the number of packets communicated per second is improved compared to previous-generation cores [20]. Multicore Navigator also enables the low latency and zero interrupts.

Texas Instruments (TI) has long been working on low power consumption for processors and that includes C66x [DSP]. Taking full advantage of TI's innovative low-power technology so called SmartReflex, C66x devices deliver additional power savings from previous-generation DSP devices. A SmartReflex device, the C66x multicore DSPs can dynamically adjust supply voltages in response to environmental conditions. In previous generations of DSPs, supply voltage would be hardwired into the device during production. But C66x devices now can constantly monitor temperatures on the chip and correlate this with the device's supply voltage. If temperatures drop, the device can automatically adjust its supply voltage downward and achieve another dimension in power savings.

Figure 3.5 shows the TMS320C6678 evaluation module where all the computations and measurements were performed.

---

1 SmartReflex technologies embed intelligence to adaptively adjust voltage, frequency and power based on device activity, modes of operation and temperature for maximum power reduction [21].
3.4 Programming environment on Digital Signal Processor

To have an optimized compiler for [DSP] is very important. The C6000 C/C++ compiler is a full-featured optimizing compiler that takes standard [ISO] C source code and converts it into C6000 assembly language source. The source code is written in C/C++ programming languages. The standard C/C++ compiler also has optimization flag that help developer to generate optimal output, assembler source is also obtained with different optimization modes. After compiler finish its work, we have the Assembler and an assembly optimizer that takes assembler source and optimizes it.
The Assembly optimizer accepts linear written assembler code that has not been register-allocated and is unscheduled. Then it assigns registers and uses loop optimization to turn the linear assembly into highly parallel assembly that takes advantage of software pipelining. After that, the Assembler translates the assembly language source files into machine language object files. In this case machine language is based on common object file format (COFF). Afterward, the Linker combines the object files into single executable object module. While creating a single executable object module, the linker also performs relocation and resolves external references such as memory allocation for parameters. After all the above steps, we finally obtain a module that can be executed in a TMS320C6000 device.

For debugging, one can use several debugging tools to refine, check and correct the code. In this occasion, we have used the USB emulator with Code Composer Studio (CCS). CCS is provided by TI together with useful scripts and tools, based on the Eclipse open source project. CCS is very helpful for programmers to manage project, external files, build linkers, connect to device, compile the source code and
debug or run the executable.

### 3.5 Sparsity, flops, and energy efficiency measures

We provide some basic terminology we use in the results chapter of this thesis:

**Sparsity**

It is referred as the percentage of nonzero elements in the sparse matrix. Typically 10% of sparsity is used for our measurements. For instance, a $100 \times 100$ matrix with 10% sparsity would have 1000 non-zero elements.

**FLOPS**

It is referred as a **F**loating-point **o**perations **p**er **s**econd. To calculate Flops we need to use a simple formula which is the number of operations in each cycle divided by time. The sparse matrix vector multiplication requires a number of operations per cycle that is two times the number of non-zeros. As we presented in the pseudo-code 3.1, there are two operations per cycle. Therefore, the number of computations is the number of non-zero elements multiplied by two and divided by the run-time:

\[
Flops = \frac{2 \cdot \text{nnz/time}}{}
\]

Equation does not include memory operations and bandwidth.

**Power Efficiency [ Flops/Watt ]**

The metric of Flops per Watt is one of the key measures of the operational cost of an installed systems. It is widely used to determine the rank of the supercomputers. The Green500 list \[24\] ranks the most energy efficient computers from the Top500 list \[24\].
Chapter 4

Implementation of sparse matrix vector multiplication

In this chapter, we discuss the different algorithms of sparse matrix vector multiplication and we provide the pseudo-code for them. Each algorithm is discussed and their implementation is described.

As we have described previously the sparse matrix storage formats (in Section 3.1.), the matrix vector multiplication can be expressed in different forms depending on the different storage formats.

4.1 Naive sparse matrix vector multiplication

In Listing 4.1 we can see the naive sparse matrix vector multiplication pseudo-code for CSR format.

Listing 4.1. Naive sparse matrix vector multiplication implementation

```c
void spmxv(int N, int indx[], int rowp[], double matvals[],
            double invec[], double outvec[]){
    int i, j;
    for (i=0;i<N;i++) {
        for (j=rowp[i]; j<rowp[i+1]; j++){
            outvec[i] += (*matvals++) *invec[*indx++];
        }
    }
}
```

where function takes \( N \) - size, \( indx \) - index pointer, \( rowp \) - row pointer (vector), \( matvals \) - matrix nonzero values, \( invec \) - input vector, as an input and outputs \( outvec \) - output vector.
4.2 Register-blocked sparse matrix vector multiplication

We have also implemented the Register-blocked naive sparse matrix vector multiplication. With different locality in matrix storage, we obtain a slightly different algorithm. The advantage of this Register-blocked sparse matrix vector multiplications is that blocks can be selected according to the register usage and size. Each small block in sparse matrix can be taken so that it fits into the registers in one computational cycle. In this way, we are able to reuse registers. Following listing 4.2 presents General algorithm for the Register-blocked Sparse matrix vector multiplication.

Listing 4.2. General implementation of Register-blocked SpMV

```c
void spmxv_regblock(int N, int r, int c, int indx[], int rowp[], double matvals[], double invec[], double outvec[])
{
    int i, j, ii, jj; // i, j for blocks, ii, jj within block
    const int num_block_rows = N/r;
    for (i=0; i<num_block_rows; i++, outvec+=r){
        for (j=rowp[i]; j<rowp[i+1]; j++, indx++, matvals+=r*c){
            for (ii =0; ii<r ; ii++)
                for (jj =0; jj<c ; jj++)
                    outvec[ii] += matvals[ii*c+jj]*invec[(indx)+jj];
        }
    }
}
```

where function input variables are N - size (row or column), r - block row size, c - block column size, indx - index pointer, rowp - row pointer (vector), matvals - matrix nonzero values, invec - input vector and output outvec - output vector.

4.3 Cache-blocked sparse matrix vector multiplication

The Cache-blocked sparse matrix vector multiplication is another method that is considered to be efficient because it can take advantage of the fast memory, namely L1 and L2 caches. The idea behind this algorithm is that the matrix and vector are divided into small blocks that fit into L1 and L2 caches. Here there is the listing for the general algorithm for Cache-blocked sparse matrix vector multiplication:

Listing 4.3. General implementation of Cache-blocked SpMV

```c
void block_spmxv(int n, int r, int indx[], int rowp[], int block_ptr[], double matvals[], double invec[], double outvec[])
{
    int i, j;
    int b_i, b_j, b_n;
```
int end_r;
// b_i=rowp[] , b_j=block_ptr[] , j=matvals[] and indx[] indices
b_n = (n+r−1)/r;
for (b_i=0; b_i<b_n; b_i++){
    end_r = (b_i+1 < b_n) ? r : n - b_i*r ;
    for (b_j=rowp[b_i]; b_j<rowp[b_i+1]; b_j+=end_r){
        for (i=0; i<end_r; i++){
            double t=0.0;
            for (j=block_ptr[b_j+i]; j<block_ptr[b_j+i+1]; j++)
                t += matvals[j] * invec[indx[j]];
            outvec[b_i*r+i] += t;
        }
    }
}

where block_ptr - pointer for the each small blocks, b_i and b_j respective indices for each small blocks.

4.4 Cache coherence on Digital Signal Processors

The numerical experiments are carried out on the TI DSP. The eight cores of the TI DSP are loosely-coupled and they do not include a mid-level caches nor provide a shared last-level cache coherency. Instead, explicit inter-core communication is used to provide core-level parallelism. Each core has its own two levels of cache. Cache can be used effectively if it is reconfigured by the program so that a portion or all of both level of caches can be used as a software controlled scratchpad memory.

Cache coherent processors keep a copy of the shared data in the local cache. Whenever a copy of the data is updated/changed, all the copies of that data will be changed as well.

An example of this type of processors is the widely used Intel processor chip, shown in Figure 4.1. The hardware has control over the data updates/changes in all-level memories. In the Intel’s Nehalem architecture, each core has its own Level 2 (L2) cache. L2 cache has better performance. Level 3 (L3) cache is used as shared memory. It duplicates L1, L2 data. This improves the inter-core communication because each core does not require to locate data from other core’s fast memory.
Figure 4.1. Example of Cache coherent processor, eight-core Nehalem [25]

Figure 4.2. Cache Coherence Problem [26] on DSP. Reproduced with kind permission by Texas Instruments Inc.
The L3 cache has flags to keep track of data and its source. When the value/data is modified in L3 cache, the data in other cores need to update the L1/L2 values with the new data to keep cache coherency.

DSPs are not a cache coherent architecture. This is shown in Figure 4.2. The cache controller provides various commands to keep coherency, but cache coherence should be manually implemented. When the data is updated in the external memory or any other memories, the updated value is kept only on that level of memory. Therefore, there will be an incorrect data and as a result the output values will not be reliable.

4.5 Parallelization of sparse matrix vector multiplication

In DSP architecture, there is no coherence between the caches. This makes parallelization difficult because very few techniques are available to solve this problem. The sparse matrix vector multiplication uses more memory operations rather than processor power for computing. This is not convenient, if there is communication between cores. This will result in a poor speed-up. We parallelize the algorithm with the following technique: the external memory Random Access Memory (RAM) is used as a cache so that all cores can access the data. By providing more memory for the data, we make sure that there will not be memory leaks or data miss. In addition, we want to avoid each core to wait for other cores to finish their job and run problem separately. To solve this issue, we allocate memory sections for each core. This will allow us to allocate memory dynamically and to run the code for eight cores.

In Chapter 6 we describe the optimization in more details.
Chapter 5

Optimizations for sparse matrix vector multiplication

This chapter provides additional details about the algorithms presented in Chapter 4, describes how to optimize them and how to improve the performance of sparse matrix vector multiplication on DSP.

5.1 Optimize Algorithms for Digital Signal Processor

We have presented the architecture of the DSP in Section 3.3. In the DSP, there are two register files named A and B. Both registers hold four functional units, \( M \) - Multiplier, \( L \) - Arithmetic Logic Unit (ALU), \( D \) - Data, \( S \) - Control. These independent functional units can provide an efficient pipelining: up to eight parallel instructions each cycle with eight core can be completed.

The optimized C compiler schedules instructions efficiently. One simple way of using registers efficiently would be to complete loop unrolling for naive algorithm 4.1.

The loop unrolling technique consists of using the same naive algorithm increasing number of operations in each iteration. We can multiply and add \( N \) rows at a time. An example of this technique is presented in Listing 5.1.

Listing 5.1. Two loops unrolling of naive SpMV

```
void spmxv_2( int nrows, int nelmts, int indx[], int rowp[],
              double matvals[], double invec[], double outvec[] ){
    int i, j;
    for ( i = 0; i < nrows-1; i++ ){
        int lb = rowp[i];
        int ub = rowp[i+1];
        int nrest = (ub-lb)%2;
        for ( j = lb; j < ub-nrest; j+=2 ){
            int index0 = indx[j];
            int index1 = indx[j+1];
```
We can change number of operation for each loop by changing the lines 7, 8 and add 12 and tune the performance of the code. Another example of loop unrolling with four operations in each loop is shown in Listing 5.2.

Listing 5.2. Four loops unrolling of naive SpMV

```c
void spmxv_4( int nrows , int nelmts , int indx [], int rowp [],
        double matvals [], double invec [], double outvec [] ){

    int i , j ;
    for ( i = 0; i < nrows - 1; i++ ){
        int lb = rowp[ i ];
        int ub = rowp[ i +1 ];
        int nrest = (ub-lb)%4;
        for( j = lb; j < ub-nrest; j+=4 ){
            int index0 = indx[ j ];
            int index1 = indx[ j +1 ];
            int index2 = indx[ j +2 ];
            int index3 = indx[ j +3 ];
            outvec[ i]=outvec[ i]+matvals[ j]*invec[ index0 ]
            +matvals[ j+1]*invec[ index1 ];
            +matvals[ j+2]*invec[ index2 ];
            +matvals[ j+3]*invec[ index3 ];
        }
    }
    for ( j = ub - nrest; j < ub; j++){
        outvec[ i] = outvec[ i] + matvals[ j]*invec[ indx[ j ]];
    }
}
```
loads data from memory more frequently. But the Register-blocked algorithm does exactly what we require. In Listing 4.2 the general Register-blocked sparse matrix multiplication with Block compressed sparse row (BCSR) storage format is shown (see Figure 3.2). One of the advantages of this format is that it reduces the indx storage by a factor of $r \times c$, where $r$ and $c$ are the size of the block. The blocked matrix-vector multiplication code becomes a series of dense $r$ by $c$ matrix-vector multiplications which allow register reuse. This algorithm can be selected, so that we can choose specific size of blocks and improve the efficiency of the algorithm. Listing 5.3 is the generated assembler code for the general Register-blocked sparse matrix vector multiplication function (Listing 4.2).

Listing 5.3. Assembly snippet for general Register-blocked function

```assembly
; * SOFTWARE PIPELINE INFORMATION
; * Resource Partition:
; * .L units 0 0
; * .S units 0 0
; * .D units 2* 2*
; * .M units 1 0
; * .X cross paths 2* 0
; * .T address paths 2* 2*

; * Searching for software pipeline schedule at ...
; * ii = 13 Schedule found with 2 iterations in parallel
; * Done

The generated assembly code shows us the information about the usage of register files (.A and .B), functional units (.L, .S - arithmetic, logical; .D - load data from memory; .M - multiply operations) and pipeline information.

It is clear from line 29 that we have 13 schedules, and that the compiler manages to complete two iterations in parallel.

After optimizing the general blocked register multiplication with $2 \times 2$ blocks in Listing 5.4 we obtain a different generated assembly code that shows some improvement in the results (see Listing 5.5). In this optimization, the inner loops are eliminated, and we improve the algorithm without small block loops, unfolded into for loops.

Listing 5.4. Register-blocked $2 \times 2$ SpMV

```
void spmxv_2x2( int nrows, int indx[], int rowp[], double matvals[], double invec[], double outvec[] )
{
    int i, j;
    const int r = 2;
    const int num_block_rows = nrows / r;
    for (i=0; i<num_block_rows; i++){
        register double d0, d1;
        d0 = outvec[2*i+0];
        d1 = outvec[2*i+1];
        for (j=rowp[i]; j<rowp[i+1]; j++){
            d0 += matvals[j*4+0]*invec[indx[j]+0];
            d0 += matvals[j*4+1]*invec[indx[j]+1];
            d1 += matvals[j*4+2]*invec[indx[j]+0];
            d1 += matvals[j*4+3]*invec[indx[j]+1];
        }
        outvec[2*i+0] = d0;
        outvec[2*i+1] = d1;
    }
}

This optimization improves the usage of the registers. In fact, we have a perfect balance in .D units (data load) and .M units (multiplication) in both A and B register files. The pipelining is effectively increased, and the number of schedules is decreased because we have more parallel iterations.

Listing 5.5. Assembly snippet for 2 × 2 Register-blocked function

;* SOFTWARE PIPELINE INFORMATION
;*
;* Resource Partition:
;*  A-side   B-side
;* .L units   0   0
;* .S units   0   0
;* .D units   4*  4*
;* .M units   2   2
;* .X cross paths   1   2
;* .T address paths 4*   3
;*
;* Searching for software pipeline schedule at ... ii = 4 Schedule found with 5 iterations in parallel
;*   Done
;*
The optimized version of algorithm is much more efficient, because we reuse registers efficiently and in parallel. By choosing a special size for block \((r \text{ and } c \text{ values})\) and changing the form of the loops, we can improve performance. But we should also be careful not to choose larger sizes of the block, so that registers are busy and there is no available register to make more pipelining or to do more operations in parallel. This can cause poor performance, because the machine cannot allocate registers and operations are postponed. From our investigation, \(4 \times 4\) size is optimal for blocks. We present results of the different experiments in Chapter 6.

Below the Listing 5.7 shows a slightly different pipeline result. We shall discuss this in details and study how this pipelining improved performance in Chapter 6.

Listing 5.6. Register-blocked 3 \times 3 SpMV

```c
void spmxv_3x3( int nrows , int indx[] , int rowp[] , double matvals[] , double invec[] , double outvec[] )
{
    /* r=c */

    int i, j;
    const int r=3;
    const int num_block_rows = nrows/r ;
    for (i=0; i<=num_block_rows; i++) {
        register double d0, d1, d2;
        d0 = outvec[3*i+0];
        d1 = outvec[3*i+1];
        d2 = outvec[3*i+2];
        for (j=rowp[i]; j<rowp[i+1]; j++)
        {
            d0 += matvals[9*j+0] * invec[indx[j]+0];
            d1 += matvals[9*j+1] * invec[indx[j]+1];
            d1 += matvals[9*j+2] * invec[indx[j]+2];
            d1 += matvals[9*j+3] * invec[indx[j]+3];
            d2 += matvals[9*j+4] * invec[indx[j]+4];
            d2 += matvals[9*j+5] * invec[indx[j]+5];
            d2 += matvals[9*j+6] * invec[indx[j]+6];
            d2 += matvals[9*j+7] * invec[indx[j]+7];
            d2 += matvals[9*j+8] * invec[indx[j]+8];
        }
        outvec[3*i+0] = d0;
        outvec[3*i+1] = d1;
        outvec[3*i+2] = d2;
    }
}
```

Another example of optimized register blocking, with three by three block size is shown in Listing 5.7.
When the compiler translates the C code into assembly code, the used registers are shown for each function. This can indicate the reuse of register files. An example of register usage can be seen in Listing 5.8.

```
Listing 5.8. Used registers for the function spmxv_3x3

;* FUNCTION NAME: spmxv_3x3
;* Local Frame Size : 0 Args + 0 Auto + 0 Save = 0 byte
```

Previously, we briefly presented the Cache-blocked sparse matrix vector multiplication in Section 4.3. Theoretically, this method should provide us higher performance. In Listing 4.3, we presented a simple implementation of the Cache-blocked SpMV. We investigate its assembly (Listing 5.9) to check if there is an efficient usage of the registers. We can notice that the pipelining gave better results than the 2 × 2 Register-blocked SpMV.
In this case, the multiplications are simply multiplication of small dense matrix vectors. Blocks are dense matrices. Therefore, this algorithm has good use of the load and multiplication registers. The block sizes can be defined according to the cache size we want to use, so that they can fit into the fast memories (L1 and L2 caches).

Listing 5.9. Assembly snippet for cache blocked SpMV

```assembly
; SOFTWARE PIPELINE INFORMATION
; Resource Partition:
; .L units 0 0
; .S units 0 0
; .D units 2* 1
; .M units 1 0
; .X cross paths 1 0
; .T address paths 2* 1
; Searching for software pipeline schedule at ...
; ii = 3 Schedule found with 5 iterations in parallel
; Done
```

Note that if we do not use L1 and L2 cache, then there is no advantage of using such a method: we perform several unnecessary multiplications with small blocks. This does not compensate the memory bandwidth performance. Because there is no cache coherency, it is very challenging to optimize the Cache-blocked sparse matrix multiplication and the memory management becomes complex.

We can see more details about these results in the following Chapter 6.

5.2 Parallel optimization

In the register blocking technique, we reorganize the matrix data structure and associated computation to improve the reuse of data in the source vector, without destroying the locality in the destination vector.

But differently from the register blocking approach, the set of values in the caches is not under complete control of the software. In fact, the hardware controls the selection of data values in each level of cache according to its policies on replacement, associativity, and write strategy. Most importantly, the caches can hold thousands of values, while the registers only hold tens of values, so it is not practical to all larger blocks of a sparse matrix, source vector and destination vector values to fit in cache at the same time [27]. Instead, we rearranged the computation so that a block of values in the matrix are accessed near each other in time, still retaining the
sparse structure of the matrix. The register blocking technique has the advantage that it does not allow indexing and loop overhead.

In our experiments, we created a memory scratchpad so that each core uses an external memory. The memory for each core does not overlap with other core memory. The software keeps track of the memory state. The drawback is that there is a large number of memory read and write operations. This is expected from sparse matrix operations. Since all core has their own data generated, results are similar. Therefore, the results are collected from one master core and they are averaged. The master core calculates the mean value and writes the output log files. In next the Chapter 6, all the results are average values taken from eight-cores, and it is essential to understand that the results of performance, energy efficiency are average values for one core.
Chapter 6

Measurements and Results

In this chapter we present the results of the measurements, for computational performance and for energy efficiency. Additional tools and scripts to collect, process and visualize data are shown.

6.1 Post-processing scripts

There is a number of additional tools that we have used to measure the power usage during our experiments. As can be seen in Table 6.5, we have four power channels to measure the energy usage. Small shunt resistors were placed at four points to measure the current delivered to major components of the evaluation module. At the same time, we measure the voltage present on the rails to be able to calculate power accurately. These sensed signals flowed to the amplifiers to convert and scale the differential input signals and remove common mode noise after a 9th order Butterworth anti-alias filter to remove high frequency components for each channel. Afterwards, the filtered signals are digitalized by a National Instruments cRIO-9074 based system using NI-9205 Analog to Digital Converter modules. The digitized data is sent via Ethernet using User Datagram Protocol (UDP) to the acquisition host. There it is time-stamped and correlated with benchmark progress information received from the DSP via a RS232 line \[28\].

After we collect the data using `udplogger`, we process it using the `AWK` scripting language.

`AWK` is a Unix command line script that helps to parse text files. It has a built-in support for many functions to post-process data for visualization purposes. Below, Listing 6.1 shows an example of a script to read a log and clear short runs.

\[\text{designed by Alfred Aho, Peter Weinberger, and Brian Kernighan}\]
Listing 6.1. Example of AWK scripting language

```
#! /usr/bin/awk -f
#
# Remove short runs from the merged or
# trimmed logs (also works on raw logs)
#
BEGIN {
    if (! cutoff ) cutoff=2
    seen_start = 0
    seen_name = 0
    good_run = 0
    FS = '\t'
    CONVFMT = "%20.6f"
```
total_runs = 0
long_runs = 0

END {
    if (good_run != 0) {
        print run_data
        long_runs += 1
    }
    print "Seen", total_runs "experiment_runs, kept"  
        long_runs > "//dev/stderr"
}

# We can see a new run starting, possible output the old one if we have one
$2 == "N" {
    if (good_run != 0) {
        print run_data
        long_runs += 1
    }
    run_data = ""
    seen_name = 1
    good_run = 0
    total_runs += 1
}

# We can see the start marker, record time
$2 == "S" {
    start_time = $1
    seen_start = 1
}

# We can see the end marker, record time and check cutoff
$2 == "E" {
    end_time = $1
    seen_end = 1
    run_len = end_time - start_time
    if (run_len > cutoff) good_run = 1
    else good_run = 0
}

    if (run_data) run_data = run_data "\n" $0
    else run_data = $0

With AWK we create parsed text files that are visualized using Gnuplot, a graphic utility. Gnuplot is an open source project supported by the scientific community. One of the advantages of this tool is that it works easily with text data
files. Gnuplot supports a number of output file extensions or formats, including most popular formats, *.jpeg, *.png, *.bmp, vector graphics formats *.svg, *.eps, etc.

6.2 Energy and Performance results

In order to get more samples of the data, we need to have measures over a period of time that is longer than the code run time. Because the code run time is short, in most cases we will not be able to get enough power samples to work with. For this reason, the resulting data may not be sufficient. To solve this problem, we increase the run time by increasing the number of iterations. When we have enough number of samples, there is a higher chance that our approximation will be more accurate.

Udplogger logs all the data. After that, the Awk script calculate channel energy parse the logs and prepare all the power samples. The raw data is analogous to the data presented in Figure 6.2. In this example, we have power values from the channels 0, 1, 2 and 3.

| 16735.895930025 | 1 | 3.83527 | 0.00082507 | 1084.24 |
| 16735.895890025 | 2 | 0.782876 | 0.000081665 | 211.638 |
| 16735.895820025 | 3 | 0.456191 | 0.000045731 | 183.477 |
| 16735.895165025 | 0 | 13.2603 | 0.0135765 | 3769.82 |
| 16735.895134025 | 1 | 3.82697 | 0.00082086 | 1084.24 |
| 16735.895124025 | 2 | 0.783524 | 0.000082636 | 211.639 |
| 16735.895155025 | 3 | 0.456569 | 0.0000467526 | 183.477 |
| 16735.895595025 | 4 | 13.2621 | 0.0135489 | 3769.83 |
| 16735.895585025 | 5 | 3.833165 | 0.00082561 | 1084.25 |
| 16735.895666025 | 6 | 0.783587 | 0.000082506 | 211.64 |
| 16735.895740025 | 3 | 0.456302 | 0.0000467253 | 183.478 |
| 16735.895740025 | 0 | 13.2696 | 0.0135574 | 3769.84 |
| 16735.895820025 | 1 | 3.82692 | 0.00082091 | 1084.25 |
| 16735.895890025 | 2 | 0.783515 | 0.0000821117 | 211.64 |
| 16735.895890025 | 3 | 0.456739 | 0.0000466979 | 183.478 |
| 16735.896290025 | 0 | 13.2607 | 0.0135565 | 3769.86 |
| 16735.896060025 | 1 | 3.833102 | 0.00082296 | 1084.25 |
| 16735.896040025 | 2 | 0.783551 | 0.000082151 | 211.641 |
| 16735.896020025 | 3 | 0.456227 | 0.0000467176 | 183.479 |
| 16735.896020025 | 0 | 13.2633 | 0.0135561 | 3769.87 |
| 16735.896250025 | 1 | 3.833101 | 0.00082235 | 1084.26 |
| 16735.896230025 | 2 | 0.783709 | 0.000082517 | 211.642 |
| 16735.896260025 | 3 | 0.456246 | 0.0000467165 | 183.479 |
| 16735.897060025 | 0 | 13.2626 | 0.0135564 | 3769.88 |
| 16735.897050025 | 1 | 3.833001 | 0.00082193 | 1084.26 |
| 16735.897060025 | 2 | 0.783566 | 0.000082151 | 211.643 |
| 16735.897070025 | 3 | 0.456742 | 0.0000467073 | 183.49 |
| 16735.898270025 | 0 | 13.2629 | 0.0135557 | 3769.9 |
| 16735.898270025 | 1 | 3.833142 | 0.00082235 | 1084.27 |
| 16735.898280025 | 2 | 0.783685 | 0.0000824288 | 211.644 |
| 16735.898080025 | 3 | 0.456245 | 0.0000467185 | 183.48 |
| 16735.898090025 | 0 | 13.2691 | 0.0135671 | 3769.91 |
| 16735.899100025 | 1 | 3.832955 | 0.00082149 | 1084.27 |
| 16735.899110025 | 2 | 0.782863 | 0.000081652 | 211.644 |

Figure 6.2. Snippet of generated raw power samples
After the data is generated, we process it using awk script. Since we have a power samples over time, we can integrate the samples over time to approximate the results. For integration, we use basic trapezoidal rule. The final result generates a small set of data where all necessary logs are present as shown in Figure 6.3.

![Figure 6.3. Snippet of processed energy samples](image)

We can then work with samples to calculate the efficiency, the performance and power usage. Using the gnuplot utility we then prepare the figures.

<table>
<thead>
<tr>
<th>Name</th>
<th>NNZ</th>
<th>Time</th>
<th>power0 total</th>
<th>power1 var</th>
<th>power2 fix</th>
<th>power3 mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmxv_n</td>
<td>1000</td>
<td>0.000048797</td>
<td>13.757656</td>
<td>4.141722</td>
<td>0.788435</td>
<td>0.464416</td>
</tr>
<tr>
<td>spmxv_n</td>
<td>4000</td>
<td>0.00018874</td>
<td>13.786841</td>
<td>4.145954</td>
<td>0.787851</td>
<td>0.46445</td>
</tr>
<tr>
<td>spmxv_n</td>
<td>16000</td>
<td>0.000741186</td>
<td>13.830329</td>
<td>4.142453</td>
<td>0.788469</td>
<td>0.464412</td>
</tr>
<tr>
<td>spmxv_n</td>
<td>36000</td>
<td>0.00165728</td>
<td>13.885631</td>
<td>4.152606</td>
<td>0.788147</td>
<td>0.494069</td>
</tr>
<tr>
<td>spmxv_n</td>
<td>100000</td>
<td>0.00460632</td>
<td>14.633221</td>
<td>4.243027</td>
<td>0.791496</td>
<td>1.033822</td>
</tr>
<tr>
<td>spmxv_n</td>
<td>400000</td>
<td>0.0183485</td>
<td>14.646213</td>
<td>4.265126</td>
<td>0.794317</td>
<td>1.03871</td>
</tr>
</tbody>
</table>

Table 6.1. Results for naive sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt.

In the tables, Name - stands for the name of the algorithms, NNZ - number of nonzero elements, Time - algorithm runtime in seconds, power0 total - power result from channel 0, total power for evaluation board, power1 var - channel 1, core variable (cache and other rail powers), power2 fix - channel 2, Core fixed (core units), power3 mem - power samples from channel 3, external memory units (RAM).

Table 6.1 shows the results for naive sparse matrix vector multiplication. Sparsity is constant 10% for all problem sizes.
<table>
<thead>
<tr>
<th>Name</th>
<th>NNZ</th>
<th>Time</th>
<th>power0 total</th>
<th>power1 var</th>
<th>power2 fix</th>
<th>power3 mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmxv_2</td>
<td>1000</td>
<td>0.000039015</td>
<td>13.729894</td>
<td>4.146546</td>
<td>0.787545</td>
<td>0.464212</td>
</tr>
<tr>
<td>spmxv_2</td>
<td>4000</td>
<td>0.000147784</td>
<td>13.728295</td>
<td>4.152798</td>
<td>0.789621</td>
<td>0.464351</td>
</tr>
<tr>
<td>spmxv_2</td>
<td>16000</td>
<td>0.00057587</td>
<td>13.685869</td>
<td>4.139992</td>
<td>0.788475</td>
<td>0.464348</td>
</tr>
<tr>
<td>spmxv_2</td>
<td>36000</td>
<td>0.00128417</td>
<td>13.77246</td>
<td>4.151276</td>
<td>0.787565</td>
<td>0.495402</td>
</tr>
<tr>
<td>spmxv_2</td>
<td>100000</td>
<td>0.00357318</td>
<td>14.674148</td>
<td>4.25329</td>
<td>0.790618</td>
<td>1.147531</td>
</tr>
<tr>
<td>spmxv_2</td>
<td>400000</td>
<td>0.0142504</td>
<td>14.756616</td>
<td>4.267455</td>
<td>0.792361</td>
<td>1.160999</td>
</tr>
</tbody>
</table>

Table 6.2. Results for two rows loop unrolled sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt.

After the optimization of the naive algorithm, the two loops unrolling results are shown in Table 6.2.

<table>
<thead>
<tr>
<th>Name</th>
<th>NNZ</th>
<th>Time</th>
<th>power0 total</th>
<th>power1 var</th>
<th>power2 fix</th>
<th>power3 mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmxv_4</td>
<td>1000</td>
<td>0.000033919</td>
<td>13.756997</td>
<td>4.159679</td>
<td>0.788995</td>
<td>0.4641</td>
</tr>
<tr>
<td>spmxv_4</td>
<td>4000</td>
<td>0.000124801</td>
<td>13.737473</td>
<td>4.153917</td>
<td>0.78948</td>
<td>0.464132</td>
</tr>
<tr>
<td>spmxv_4</td>
<td>16000</td>
<td>0.000477746</td>
<td>13.728193</td>
<td>4.160889</td>
<td>0.789791</td>
<td>0.46416</td>
</tr>
<tr>
<td>spmxv_4</td>
<td>36000</td>
<td>0.00106089</td>
<td>13.826649</td>
<td>4.170645</td>
<td>0.789522</td>
<td>0.505736</td>
</tr>
<tr>
<td>spmxv_4</td>
<td>100000</td>
<td>0.00294129</td>
<td>14.898504</td>
<td>4.298196</td>
<td>0.793811</td>
<td>1.288263</td>
</tr>
<tr>
<td>spmxv_4</td>
<td>400000</td>
<td>0.0117015</td>
<td>14.979474</td>
<td>4.316916</td>
<td>0.795812</td>
<td>1.326669</td>
</tr>
</tbody>
</table>

Table 6.3. Results for four rows loop unrolled sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt.

Table 6.3 presents the results from four loop unrolling multiplication. A slight improvement on the results can also be seen: the execution time decreases but not significantly. Increasing number of unrolls, in Table 6.4, the improvement becomes negligible. For test purpose, a 10 loop unrolling was implemented. However, performance results were low, when compared to the two loop unrolling results.

<table>
<thead>
<tr>
<th>Name</th>
<th>NNZ</th>
<th>Time</th>
<th>power0 total</th>
<th>power1 var</th>
<th>power2 fix</th>
<th>power3 mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmxv_6</td>
<td>1000</td>
<td>0.000037161</td>
<td>13.62689</td>
<td>4.11461</td>
<td>0.78021</td>
<td>0.46413</td>
</tr>
<tr>
<td>spmxv_6</td>
<td>4000</td>
<td>0.000127276</td>
<td>13.743262</td>
<td>4.121208</td>
<td>0.783392</td>
<td>0.464325</td>
</tr>
<tr>
<td>spmxv_6</td>
<td>16000</td>
<td>0.000466318</td>
<td>13.72168</td>
<td>4.140913</td>
<td>0.785445</td>
<td>0.464242</td>
</tr>
<tr>
<td>spmxv_6</td>
<td>36000</td>
<td>0.00101929</td>
<td>13.734424</td>
<td>4.154111</td>
<td>0.785564</td>
<td>0.464247</td>
</tr>
<tr>
<td>spmxv_6</td>
<td>100000</td>
<td>0.00281039</td>
<td>14.978832</td>
<td>4.28918</td>
<td>0.790643</td>
<td>1.380405</td>
</tr>
<tr>
<td>spmxv_6</td>
<td>400000</td>
<td>0.0110882</td>
<td>15.072341</td>
<td>4.325149</td>
<td>0.796705</td>
<td>1.40401</td>
</tr>
</tbody>
</table>

Table 6.4. Results for six rows loop unrolled sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt.
In Table 6.5, we present the results about the Register-blocked sparse matrix vector multiplication with block size of $2 \times 2$. The number of non-zeros were generated from a three-point stencil in 2D, sparse matrix.

<table>
<thead>
<tr>
<th>Name</th>
<th>NNZ</th>
<th>Time</th>
<th>power0 total</th>
<th>power1 var</th>
<th>power2 fix</th>
<th>power3 mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmxv_2x2</td>
<td>592</td>
<td>0.000010341</td>
<td>13.809432</td>
<td>4.12281</td>
<td>0.786806</td>
<td>0.464201</td>
</tr>
<tr>
<td>spmxv_2x2</td>
<td>1192</td>
<td>0.00002071</td>
<td>13.803384</td>
<td>4.119485</td>
<td>0.782943</td>
<td>0.464171</td>
</tr>
<tr>
<td>spmxv_2x2</td>
<td>2392</td>
<td>0.000041436</td>
<td>13.809889</td>
<td>4.11949</td>
<td>0.782989</td>
<td>0.464346</td>
</tr>
<tr>
<td>spmxv_2x2</td>
<td>3592</td>
<td>0.000062185</td>
<td>13.814214</td>
<td>4.12626</td>
<td>0.783112</td>
<td>0.464421</td>
</tr>
<tr>
<td>spmxv_2x2</td>
<td>5992</td>
<td>0.00010367</td>
<td>13.818953</td>
<td>4.126191</td>
<td>0.782577</td>
<td>0.464286</td>
</tr>
</tbody>
</table>

Table 6.5. Results for Register-blocked sparse matrix vector multiplication with $2 \times 2$. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt.

The increase of the block sizes improved the performance, as we can see in Table 6.6 and in Table 6.7. The results prove that the optimizations we have presented in Section 5.1 are effective.

<table>
<thead>
<tr>
<th>Name</th>
<th>NNZ</th>
<th>Time</th>
<th>power0 total</th>
<th>power1 var</th>
<th>power2 fix</th>
<th>power3 mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmxv_3x3</td>
<td>882</td>
<td>0.000009373</td>
<td>13.982733</td>
<td>4.262348</td>
<td>0.788174</td>
<td>0.464341</td>
</tr>
<tr>
<td>spmxv_3x3</td>
<td>1782</td>
<td>0.000018745</td>
<td>13.990455</td>
<td>4.258189</td>
<td>0.788208</td>
<td>0.464273</td>
</tr>
<tr>
<td>spmxv_3x3</td>
<td>3582</td>
<td>0.000037774</td>
<td>13.985727</td>
<td>4.261804</td>
<td>0.788314</td>
<td>0.464364</td>
</tr>
<tr>
<td>spmxv_3x3</td>
<td>5382</td>
<td>0.000056909</td>
<td>13.991103</td>
<td>4.262941</td>
<td>0.788002</td>
<td>0.464438</td>
</tr>
<tr>
<td>spmxv_3x3</td>
<td>8982</td>
<td>0.000094787</td>
<td>14.015081</td>
<td>4.278327</td>
<td>0.789857</td>
<td>0.464423</td>
</tr>
<tr>
<td>spmxv_3x3</td>
<td>17982</td>
<td>0.000189305</td>
<td>14.030373</td>
<td>4.275731</td>
<td>0.78055</td>
<td>0.464334</td>
</tr>
</tbody>
</table>

Table 6.6. Results for Register-blocked sparse matrix vector multiplication with $3 \times 3$. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt.

<table>
<thead>
<tr>
<th>Name</th>
<th>NNZ</th>
<th>Time</th>
<th>power0 total</th>
<th>power1 var</th>
<th>power2 fix</th>
<th>power3 mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>spmxv_4x4</td>
<td>1168</td>
<td>0.000010415</td>
<td>14.056656</td>
<td>4.322882</td>
<td>0.787141</td>
<td>0.464321</td>
</tr>
<tr>
<td>spmxv_4x4</td>
<td>2368</td>
<td>0.000020895</td>
<td>14.07082</td>
<td>4.329258</td>
<td>0.787476</td>
<td>0.464402</td>
</tr>
<tr>
<td>spmxv_4x4</td>
<td>4768</td>
<td>0.000041862</td>
<td>14.060876</td>
<td>4.322927</td>
<td>0.788096</td>
<td>0.464241</td>
</tr>
<tr>
<td>spmxv_4x4</td>
<td>7168</td>
<td>0.000062838</td>
<td>14.077148</td>
<td>4.335499</td>
<td>0.787603</td>
<td>0.464209</td>
</tr>
<tr>
<td>spmxv_4x4</td>
<td>11968</td>
<td>0.000104743</td>
<td>14.073639</td>
<td>4.329355</td>
<td>0.787924</td>
<td>0.464293</td>
</tr>
<tr>
<td>spmxv_4x4</td>
<td>23968</td>
<td>0.000209545</td>
<td>14.064081</td>
<td>4.332081</td>
<td>0.787375</td>
<td>0.464259</td>
</tr>
</tbody>
</table>

Table 6.7. Results for Register-blocked sparse matrix vector multiplication with $4 \times 4$. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt.
Table 6.8. Results for Register-blocked sparse matrix vector multiplication with $10 \times 10$. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt.

Table 6.8 shows the results for the Register-blocked matrix sparse vector multiplication with block size of $10 \times 10$. As we have discussed previously (in Section 5.1), increasing size of blocks beyond $10 \times 10$ did not show any further improvement on performance and power consumption. This is because there were not enough registers to allocate or use.

Table 6.9. Results for general Cache-blocked sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt.

Table 6.9 shows the results for the Cache-blocked sparse matrix vector multiplication. Although the power usage remains low, the computational performance of the Cache-blocked multiplication is low with our technique.

Now we can visualize the data, by using the gnuplot utility. We have following plots. In Figure 6.4, we can see the performance of all the algorithms. The sparse matrix vector multiplication does not perform well on many architectures because of its dependence on indirect addressing and because of the several memory accesses per operation. For this reason, sparse matrix vector multiplication has lower chance to benefit from [SIMD] instructions. The results show that Register-blocked algorithm gives the highest performance, while the Cache-blocked has lower performance. With a $200 \times 200$ matrix size, the performance of the Cache-blocked sparse matrix vector multiplication is 14 MFlops, while the performance of the Register-blocked sparse matrix vector multiplication with block size of $4 \times 4$ is 227 MFlops.

In the Register-blocked technique larger block sizes cannot be allocated into the registers. Therefore, performance drops when the size of the problem increases,
algorithm spmv_10x10 can be seen from the Figure 6.4. Performance from loop-unrolling technique has better results compare to naive algorithm. Register-blocked sparse matrix vector multiplication highly noticeable.

For calculating the energy efficiency of each algorithm, we take the number of operations and divide it by total power usage. Our results indicate that increasing the number of non-zeros and the size of the problem, the efficiency increases linearly with different algorithms (see in Figure 6.5).

The register-blocked sparse matrix vector algorithms have higher efficiency. Efficiency increase by factor of 4 from naive to Register-blocked algorithm. Though the increase of the number of non-zeros did not affect the efficiency for each algorithm. In the case of the Cache-blocked algorithm, efficiency decreases when increasing the number of non-zeros. The reason is that cache blocked algorithm has more memory indirect accesses (read and write operations) which in result we saw performance drop as well as higher power usage.

Naive algorithm, Loop-unrolling, Cache-blocked sparse matrix vector multiplication techniques used with randomly generated sparse matrix with 10% of sparsity
Figure 6.5. Energy Efficiency of the Loop unrolling, Register and Cache blocked SpMV algorithms, changing the number of non-zeros (10% matrix values are non-zero). Register-blocked algorithm uses 3-point stencil in 2D sparse matrix which has a special structure. Hence, x-axis of the figure made with number of nonzero elements. Despite the fact that algorithms use different sparse matrices, it is clear that with same number of non-zero elements in a matrix, Register-blocked surpasses other algorithms.

Figure 6.6 shows the average power usage for the DSP eight cores during the computations. We can see that naive, Register-blocked and Cache-blocked algorithms have average power usage of 14 W. By increasing problem size, loop-unrolling technique use more power. Average power usage for loop unrolling with $1000 \times 1000$ and bigger problem size sparse matrix is $\approx 15 W$. 
Figure 6.6. Power usage in Watt for the Loop unrolling, Register and Cache-blocked SpMV algorithms, changing the problem size
Chapter 7

Discussion and Conclusions

7.1 Serial Implementation

We discussed the DSP cache coherency in Section 4.4. The non-coherent architecture is very convenient to achieve higher performance and lower energy consumption. However, it is very difficult for developers to handle memory hierarchy manually. At the beginning of thesis work, time was spent to optimize the memory usage. However, because of the limited documentation availability, it was difficult to implement specific code to optimize memory access during this thesis work period. Sparse matrix operations largely depend on the memory bandwidth, indirect addressing and arithmetic address computations. Therefore DSP has a low chance of benefiting from SIMD. To develop a Cache-blocked sparse matrix vector multiplication algorithm is very tricky because of the non coherent DSP architecture. In fact, a small change to the algorithm could result in errors in output vector. One has to be very cautious with memory boundaries.

7.2 Parallel Implementation

The communication of data between cores is an expensive operation, while the computation is relatively cheap [29]. Changing structure of the sparse matrices and improving different algorithms helped to gain higher performance. To use higher level memory still results slow. However, TI implemented special protocols, the so called MessageQ (Message Passing Interface) communication of data between cores. Developers can create a virtual heap memory that can be accessed by all cores. This technique is more difficult to be implemented because there are a limited number of documents about how this protocols work. To support on-chip memories, the DSP contains an integrated Direct memory access (DMA) which allows the device to transfer or exchange data with external memories while computations are performed. The DMA can also be programmed. This helps to perform memory flow effectively to gain from memory bandwidth. One could measure the gain from using DMA controller in terms of reduction of computation time, but there was no
chance to measure the memory bandwidth during this thesis work.

### 7.3 Energy Efficiency

The energy usage for all the algorithms was low (on average 14 W). We report the improvement of the power efficiency. For the Register-blocked we observed an improvement of a factor of four when compared to the naive algorithm power efficiency. The power efficiency of sparse matrix vector multiplication on DSP was higher than the power efficiency of other common processor architectures. For instance, the DSP with clock frequency of 1.5 GHz had an energy efficiency about three times higher for optimized HPL benchmarks than x86 platforms with clock frequencies in the 2.5-3.9 GHz range [28].

### 7.4 Conclusions

The rapid improvement of processor architectures puts increasing pressure on the energy usage and the memory access for the programs. A major problem is that technology improves the speed of logic much more rapidly than speed of the memory operations. The modern Central Processor Unit (CPU) is running very fast and data needs to be fed constantly into it. Memory is not as fast as the logic, so one has to put data in fast memories known as caches and cache controllers. Various attempts have been made to change processors’ architectures to make memory more accessible and faster as well as to decrease energy consumption.

In this thesis, we studied an eight core TI DSP architecture, its performance and its energy consumption. By implementing sparse matrix vector multiplication algorithms and optimizing them, we were able to choose the algorithms to use the instruction sets of the architectures more effectively. We were able to show that different sparse matrix structure can be used with various optimization techniques to overcome the architecture limitations.

The first part of the study included the presentation of the naive algorithm and of the loop unrolling technique. By changing the original algorithm, we were able to use instruction files and memory efficiently to allow the compiler to make use of pipelining on the low level. There was a 0.3 W power saving for smaller matrices. Because of the frequent memory accesses with bigger matrices, the power usage increased by 1.2 W than the average usage.

The second optimization technique was to improve the instructions files usage on DSP by having smaller blocks of the big sparse matrix. The Register-blocked sparse matrix vector multiplication used all the units of the instruction files effectively and pipelining was successfully accomplished. We were able to analyze algorithm from assembly extraction of the source code. Two by two, three by three, four by four and 10 by 10 block sizes of the Register-blocked sparse matrix vector multiplications were experimented. As the size of blocks increased, we gain improvement on performance. However, with 10 by 10 block size, the registers were too busy and no scheduling
was found for computation and pipelining. As a result, the performance decreased. The optimal size of the block is four by four for Register-blocked sparse matrix vector multiplication.

The third algorithm was a Cache-blocked sparse matrix vector multiplication, where matrix structure changed into small dense block matrices, similar to register blocked structure. The block sizes can be larger and can fit into cache instead of registers. Small dense matrix vector multiplications are known to be faster than sparse matrix vector multiplications. By using small dense matrix blocks from sparse matrix, one can optimize the cache usage. Since the [DSP] architecture has no cache coherence, our manual memory management techniques did not show considerable performance improvement. In theory, the cache blocked sparse matrix vector multiplication algorithm should give higher performance. However, we gained a small performance increase (less than 50 MFlops and average energy usage of 13.9 W). In conclusion, we found that the Texas Instruments [DSP] C6678 overall delivers low power consumption and good performance when Register-blocked sparse matrix vector multiplication technique is used.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Example of Compressed Sparse Row format</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of Block Compressed Sparse Row format with $2 \times 2$ dense block</td>
<td>8</td>
</tr>
<tr>
<td>3.3</td>
<td>Example of Cache-Blocked sparse matrix storage</td>
<td>9</td>
</tr>
<tr>
<td>3.4</td>
<td>Functional Block Diagram of the Evaluation Module [18], TMS320C6678</td>
<td>11</td>
</tr>
<tr>
<td>3.5</td>
<td>TMS320C6678L Evaluation module, used in this thesis</td>
<td>12</td>
</tr>
<tr>
<td>3.6</td>
<td>TMS320C6678L Evaluation module with details. Reproduced with kind permission by Texas Instruments Inc.</td>
<td>13</td>
</tr>
<tr>
<td>3.7</td>
<td>TMS320C6000 Software Development Flow [22]. Reproduced with kind permission by Texas Instruments Inc.</td>
<td>14</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of Cache coherent processor, eight-core Nehalem [25]</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Cache Coherence Problem [26] on DSP. Reproduced with kind permission by Texas Instruments Inc.</td>
<td>20</td>
</tr>
<tr>
<td>5.1</td>
<td>Parallelization Overview</td>
<td>30</td>
</tr>
<tr>
<td>6.1</td>
<td>Experiment setup and connections of measurement devices</td>
<td>32</td>
</tr>
<tr>
<td>6.2</td>
<td>Snippet of generated raw power samples</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>Snippet of processed energy samples</td>
<td>35</td>
</tr>
<tr>
<td>6.4</td>
<td>Performance of the Loop unrolling, Register and Cache blocked SpMV algorithms, changing the problem size</td>
<td>39</td>
</tr>
<tr>
<td>6.5</td>
<td>Energy Efficiency of the Loop unrolling, Register and Cache blocked SpMV algorithms, changing the number of non-zeros</td>
<td>40</td>
</tr>
<tr>
<td>6.6</td>
<td>Power usage in Watt for the Loop unrolling, Register and Cache-blocked SpMV algorithms, changing the problem size</td>
<td>41</td>
</tr>
</tbody>
</table>
## List of Tables

1.1 Clock frequency and power details for different architectures. . . . . . . 3

6.1 Results for naive sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt. . . . . . . 35

6.2 Results for two rows loop unrolled sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt. 36

6.3 Results for four rows loop unrolled sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt. 36

6.4 Results for six rows loop unrolled sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt. 36

6.5 Results for Register-blocked sparse matrix vector multiplication with $2 \times 2$. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt. 37

6.6 Results for Register-blocked sparse matrix vector multiplication with $3 \times 3$. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt. 37

6.7 Results for Register-blocked sparse matrix vector multiplication with $4 \times 4$. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt. 37

6.8 Results for Register-blocked sparse matrix vector multiplication with $10 \times 10$. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt. 38

6.9 Results for general Cache-blocked sparse matrix vector multiplication. Name - name of the algorithm; NNZ - number of non-zero elements in a matrix; Time - runtime of the algorithm in seconds; powerX - channel power values in Watt. 38
# List of Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACP</td>
<td>Average CPU Power</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>AMD</td>
<td>Advanced Micro Devices</td>
</tr>
<tr>
<td>BCSR</td>
<td>Block Compressed Sparse Row</td>
</tr>
<tr>
<td>BLAS</td>
<td>Basic Linear Algebra Subprograms</td>
</tr>
<tr>
<td>CCS</td>
<td>Code Composer Studio</td>
</tr>
<tr>
<td>COFF</td>
<td>Common Object File Format</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processor Unit</td>
</tr>
<tr>
<td>CSR</td>
<td>Compressed Sparse Row</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Programming Units</td>
</tr>
<tr>
<td>HPC</td>
<td>High Performance Computing</td>
</tr>
<tr>
<td>IBM</td>
<td>International Business Machines</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>MSM</td>
<td>Multicore Shared Memory</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>SpMV</td>
<td>Sparse Matrix Vector multiplication</td>
</tr>
<tr>
<td>TDP</td>
<td>Thermal Design Power</td>
</tr>
<tr>
<td>TI</td>
<td>Texas Instruments</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Long Instruction Word</td>
</tr>
</tbody>
</table>
Bibliography


## Appendix A: Code listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Naive sparse matrix vector multiplication implementation</td>
<td>17</td>
</tr>
<tr>
<td>4.2 General implementation of Register-blocked SpMV</td>
<td>18</td>
</tr>
<tr>
<td>4.3 General implementation of Cache-blocked SpMV</td>
<td>18</td>
</tr>
<tr>
<td>5.1 Two loops unrolling of naive SpMV</td>
<td>23</td>
</tr>
<tr>
<td>5.2 Four loops unrolling of naive SpMV</td>
<td>24</td>
</tr>
<tr>
<td>5.3 Assembly snippet for general Register-blocked function</td>
<td>25</td>
</tr>
<tr>
<td>5.4 Register-blocked $2 \times 2$ SpMV</td>
<td>25</td>
</tr>
<tr>
<td>5.5 Assembly snippet for $2 \times 2$ Register-blocked function</td>
<td>26</td>
</tr>
<tr>
<td>5.6 Register-blocked $3 \times 3$ SpMV</td>
<td>27</td>
</tr>
<tr>
<td>5.7 Assembly snippet for $3 \times 3$ Register-blocked function</td>
<td>28</td>
</tr>
<tr>
<td>5.8 Used registers for the function spmxv $_3 \times 3$</td>
<td>28</td>
</tr>
<tr>
<td>5.9 Assembly snippet for cache blocked SpMV</td>
<td>29</td>
</tr>
<tr>
<td>6.1 Example of AWK scripting language</td>
<td>31</td>
</tr>
</tbody>
</table>