Parallel Solution of the Subset-sum Problem:
An Empirical Study

A Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Master of Science in the
Graduate School of The Ohio State University

By

Saniyah S. Bokhari, B.S.

Graduate Program in
Department of Computer Science and Engineering

The Ohio State University

2011

Master's Examination Committee:

Ten Hwang Lai, Advisor

Dong Xuan

© Copyright by
Saniyah S. Bokhari
2011

Abstract

We investigate the parallelization of an algorithm on three very different architectures. These are: a 128-processor Cray XMT massively multithreaded machine, a 16-processor IBM x3755 shared memory machine and a 240-core NVIDIA FX 5800 graphics processor unit (GPU). The problem we use in our investigation is the well-known subset-sum problem. While this is known to be NP-complete, it is solvable in pseudo-polynomial time, i.e., time proportional to the number of input objects multiplied by the sum of their sizes. This product defines the size of the dynamic programming table used to solve the problem.

The hypothesis that we wish to test is that the Cray, with its specialized hardware and large uniform shared memory, is suitable for very large problems, the IBM x3755 is suitable for intermediate sized problems and the NVIDIA FX 5800 can give superior performance only for problems that fit within its modest internal memory.

We show that it is straightforward to parallelize this algorithm on the Cray XMT primarily because of the word-level locking that is available on this architecture. For the other two machines we present an alternating word algorithm that can implement an efficient solution. The timings of our respective codes were carefully measured over a comprehensive range of problem sizes. On the Cray XMT we observe very good scaling for large problems and see sustained performance as the problem size increases. However this machine has poor scaling for small problem sizes; it performs best for

problem sizes of 10^{12} bits or more. The IBM x3755 performs very well on medium sized problems, but has poor scalability as the number of processors increases and is unable to sustain performance as the problem size increases. This machine tends to saturate for problem sizes of 10^{11} bits. The NVIDIA GPU performs well for problems whose tables fit within its 4GB device memory. This corresponds to tables of size approximately 10^{10} .

The experimental measurements support our hypothesis and show that each machine gives superior performance for distinct ranges of problem sizes and demonstrate the strengths and weaknesses of the three architectures.

To Ami, Abboo and Saba

Acknowledgments

I thank my advisor, Professor Ten Hwang Lai, for his encouragement of this work. Access to the Cray XMT was provided by the Center for Adaptive Supercomputing Software–Multithreaded Architectures (CASS-MT) hosted at the Pacific Northwest National Laboratory. I thank John Feo for his support and Jace Mogill for valuable advice. This work was supported in part by an allocation of computing time from the Ohio Supercomputer Center. I am grateful to Barbara Woodall for her generous assistance.

Vita

June 15, 1987Born – Lahore, Pakistan

Sep 2006 – Aug 2009 Bachelor of Science
Computer Science & Engineering
The Ohio State University
Columbus, Ohio

Fields of Study

Major Field: Computer Science & Engineering

Table of Contents

	Page
Abstract	ii
Dedication	iv
Acknowledgments	v
Vita	vi
List of Tables	ix
List of Figures	x
1. Introduction	1
1.1 The Three Architectures	1
1.2 Subset-sum Problem	3
1.2.1 An Example	3
1.2.2 Complexity	4
1.2.3 NP-Completeness	4
1.3 A Dynamic Programming Algorithm	4
1.3.1 Complexity and Pseudo-Polynomiality	5
2. Implementation	8
2.1 Basic Implementation	8
2.2 Parallelizing for the Cray XMT	9
2.3 Parallelization on Conventional Shared Memory Machines	12

3.	Word Oriented Approach	14
3.1	Discussion	14
3.2	Alternating Word Code for Shared Memory Machines	15
3.3	Word Code for Cray XMT	15
3.4	Alternating Word Approach on NVIDIA FX 5800	20
3.5	Comparison of Results	26
3.5.1	IBM x3755	26
3.5.2	Cray XMT	27
3.5.3	NVIDIA FX 5800	27
4.	Conclusions	28
	Appendices	30
A.	Predicted times for the XMT using the Bit Oriented Approach	30
B.	Predicted times for the XMT with Word Approach	34
C.	Scheduling Policies on the XMT	37
D.	Out of Core NVIDIA code	39
	Bibliography	42

List of Tables

Table	Page
1.1 Dynamic Programming Table for solving an instance of the subset-sum problem, adapted from Garey and Johnson [5, Fig. 4.8, p. 91].	6
1.2 Backtracking in the Dynamic programming table to find the solution.	7
C.1 XMT 128 processor, 5.75×10^{12} bit problem under varying scheduling schemes.	38

List of Figures

Figure	Page
2.1 Parallelizing the subset-sum problem on the Cray XMT using a bit oriented approach	11
3.1 Any bits marked T (shaded) in row $i-1$ will set bits that are <code>size[i]</code> bits away in row i . The bits in any word in row $i-1$ can influence bits in two words in row i	16
3.2 The T bits in any word in row $i-1$ can influence <i>at most</i> two adjacent words in row i	17
3.3 Alternating Word Approach implemented using OMP on the IBM x3755	18
3.4 Word Approach implemented on the Cray XMT	19
3.5 Parallelizing the subset-sum problem on the Cray XMT using the word oriented approach	21
3.6 Performance of IBM x3755	22
3.7 Performance of NVIDIA FX 5800 using two different codes	22
3.8 Performance of all three machines	23
3.9 CUDA Kernel for copying an entire row	24
3.10 CUDA Kernel implementing Alternating Word Approach	25
3.11 Calling CUDA Kernels in the Main Program	25

Chapter 1: Introduction

In this thesis we explore the parallelization of the subset-sum problem on three contemporary but very different architectures. The subset-sum problem takes as input a list of integers and is required to determine if a subset of these integers adds up to exactly half the sum of the entire list. This problem is NP-complete but a pseudo-polynomial dynamic programming algorithm for its solution is known. We provide in this thesis the results of detailed experiments on variously sized subset-sum problems when executed on three target architectures. These experiments highlight the strengths and weakness of these architectures in the context of a well defined combinatorial problem. The hypothesis that we wish to test is that the Cray, with its specialized hardware and large uniform shared memory, is suitable for very large problems, the IBM x3755 is suitable for intermediate sized problems and the NVIDIA FX 5800 can give superior performance only for problems that fit within its modest internal memory.

1.1 The Three Architectures

The three different architectures that we evaluate parallelism on are:

1. The Cray Extreme Multithreading (XMT) supercomputer, originally developed by Tera [1, 2] which then progressed into the Multithreaded Architecture, MTA

[12] and forthwith became the current XMT [3]. It has special hardware that can support 128 threads per processor and switching between these threads requires no overhead. The availability of a powerful interconnect allows uniform access to shared memory. This machine has special full/empty bits with individual 64-bit words; these allow for word-level locking/unlocking. It is equipped with a powerful parallelizing compiler that is also accompanied by a complementary performance analysis tool, CANAL. The Center for Adaptive Supercomputer Software (CASS) (cass-mt.pnl.gov), at Pacific Northeast Laboratory (PNL) provided the large XMT machine that we used. This has 128 processors and 1 Terabyte of memory.

2. A 16 processor IBM System x3755 from the Ohio Supercomputer Center's 'Glenn' cluster (www.osc.edu). We use this architecture as our standard conventional shared memory machine. This machine has quad socket, quad core 2.4 GHz Opteron with a 64 GB RAM.
3. An NVIDIA Quadro FX 5800 GPGPU (General Purpose Graphics Processing Unit) from the Ohio Supercomputer Center's 'Glenn' cluster (www.osc.edu/supercomputing/computing/gpgpu/index.shtml). This machine has 4GB RAM, 240 cores and is connected to a host with 24GB RAM. This GPGPU is essentially designed for developing high-definition graphics and games, by harnessing its immense power of parallelism. Since there is an enormous market for such graphics application, GPUs have become increasingly inexpensive, making them attractive to use for parallel processing [4, 9, 10, 11].

1.2 Subset-sum Problem

The variant of the subset-sum problem that this thesis explores [5] takes as input a set of objects $A = \{a_1, a_2, \dots, a_n\}$ with $|A| = n$, each object a_i having a non-negative magnitude or ‘weight’ s_i . The sum of all weights is given by $\mathcal{S} = \sum_{i=1}^n s_i$. The problem is to establish if there exists a subset $H \subset A$ with $|H| = m$ such that $\mathcal{H} = \sum_{i=1}^m h_i = \frac{1}{2}\mathcal{S}$, where h_i is the weight of the i th object in H . Note that $\mathcal{S}$ must be even for there to be a solution.

The subset-sum problem is a variant of the 0–1 knapsack problem. It can be reduced to this variant by observing that the weight w_i of each object a_i is equivalent to the value v_i and that $\mathcal{H} = \frac{1}{2}\mathcal{S} = \mathcal{W}$ where $\mathcal{W}$ is the weight limit of the knapsack. In this case we desire a subset of objects H whose weight $\mathcal{H} = \sum_{i=1}^m h_i = \mathcal{W}$, where h_i is the weight of the i th object, as before.

1.2.1 An Example

To understand the subset-sum problem better, we present a simple example taken from [5, p. 91]. Suppose $A = \{a_1, a_2, a_3, a_4, a_5\}$. Let $s_1 = 1$, $s_2 = 9$, $s_3 = 5$, $s_4 = 3$, and $s_5 = 8$. Thus $\mathcal{S} = \sum_{i=1}^5 s_i = 26$ and $\frac{1}{2}\mathcal{S} = 13$. We need to find $H \subset A$ with $|H| = m$ such that $\mathcal{H} = \sum_{i=1}^m h_i = \frac{1}{2}\mathcal{S}$. We can look at the weights of the objects and determine which of these sum up to 13. One possible subset would be $h_1 = a_1$, $h_2 = a_2$ and $h_3 = a_4$, where $s_1 = 1$, $s_2 = 9$, and $s_4 = 3$. Yet another subset for this problem would have $h_1 = a_3$ and $h_2 = a_5$, where $s_3 = 5$ and $s_5 = 8$. Both these subsets are correct solutions for this particular example.

1.2.2 Complexity

One possible way of solving the subset-sum problem would be to enumerate all possible partitions of A and for each partition check if the sum of the weights is $\frac{1}{2}\mathcal{S}$. If we have n objects, there will be 2^n partitions, and each will require n summation steps. This will take $n2^n$ time. Thus the complexity of this approach is exponential. Horowitz and Sahni [6] proposed a better solution which takes $2^{n/2}$ time which, however, is still exponential.

1.2.3 NP-Completeness

The subset-sum problem is known to be NP-complete [8]. However there exists a pseudo-polynomial algorithm that can solve this problem in acceptable time. By pseudo-polynomial we mean an algorithm that is proportional to a polynomial that includes the magnitude of the input numbers (in this case half the sum of the given integers, $\frac{1}{2}\mathcal{S}$) rather than purely the number of objects (i.e., the number of integers, n).

1.3 A Dynamic Programming Algorithm

We now show how dynamic programming may be used to solve the subset-sum problem in pseudo-polynomial time. This material is based on [5]. The dynamic programming table for the example given in Section 1.2.1 is shown in Fig. 1.1. If $\mathcal{S} = \sum_{i=1}^n s_i$ is odd then the table is not made as no solution exists.

Rows in the table represent the elements of A . Thus row i represents element a_i with weight s_i . The columns of the table represent the integer sum of weights of objects in a selected subset. These range from 0 to $\mathcal{H} = \frac{1}{2}\mathcal{S}$.

We initialize this two dimensional array to 0 or F for false values. We fill the table row by row, by determining which indices will be true, based on the following relationship.

$$\begin{aligned}
t(i, j) &= \text{T} \iff \{j = 0 \vee j = s_i\}, i = 1 \\
&= \text{T} \iff \{\{t(i-1, j) = \text{T}\} \vee \{s_i \leq j\}\} \wedge \{t(i-1, j-s_i) = \text{T}\}, i > 1
\end{aligned}$$

After the table has been completely filled in, we can examine the entry $t(n, \mathcal{H})$. If this is T then there exists a solution. In order to obtain the specific subset $H \subset A$ of elements that lead to this solution, we need to perform backtracking on the table as described below.

In the process of backtracking, we start by checking to see if the last computed cell at the bottom right of the table is 1. If not then no solution exists. Assuming that this 1 lies in cell (i, j) , we move upwards in column j until we encounter the last 1 in column j . Say this is in row k . We now travel horizontally to the left by an amount `size[k]`, ending up in cell $(k, j - \text{size}[\mathbf{k}])$. The solution includes item $\mathbf{a}[\mathbf{k}]$, which is printed out. This current cell is the new (i, j) and the process is repeated until we reach the top left of the table. This process is illustrated in Table 1.2. Note that an object with index 0 and weight 0 has been added for computational convenience.

1.3.1 Complexity and Pseudo-Polynomiality

The time required to fill up the dynamic programming table is equal to the number of objects, n , times half the sum of the object weights $\mathcal{H}$. This time dominates the time for solution as the backtracking time, described above, is linear in n . The total time is $\Theta(n\mathcal{H})$ which is a pseudo-polynomial complexity as it depends on the sum of magnitudes $\mathcal{H}$. It would have been purely polynomial if it depended only on n .

Table 1.1: Dynamic Programming Table for solving an instance of the subset-sum problem, adapted from Garey and Johnson [5, Fig. 4.8, p. 91].

Sum	Weight	j \ i		0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1	1	0	T	T	F	F	F	F	F	F	F	F	F	F	F	F
10	9	2	T	T	F	F	F	F	F	F	F	F	T	T	F	F	F
15	5	3	T	T	F	F	F	F	T	T	F	F	T	T	F	F	F
18	3	4	T	T	F	T	T	T	T	T	F	T	T	T	F	T	T
26	8	5	T	T	F	T	T	T	T	T	F	T	T	T	T	T	T

Table 1.2: Backtracking in the Dynamic programming table to find the solution.

Sum	Weight	j \ i		0	1	2	3	4	5	6	7	8	9	10	11	12	13
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
10	9	2	1	1	1	0	0	0	0	0	0	0	1	1	0	0	0
15	5	3	1	1	1	0	0	0	1	1	0	0	1	1	0	0	0
18	3	4	1	1	1	0	1	1	1	1	0	1	1	1	0	1	1
26	8	5	1	1	1	0	1	1	1	1	0	1	1	1	1	1	1

Chapter 2: Implementation

In this Chapter we discuss the basic implementation of the subset-sum algorithm which produces the dynamic programming tables. We show how this simple implementation can be easily converted to a parallel program for the Cray XMT, to find solutions in parallel. In doing so we demonstrate that it is not so easily parallelizable for a conventional shared memory machine, due to certain limitations of the architecture.

2.1 Basic Implementation

It is straightforward to implement the dynamic programming algorithm using a 2-dimensional array for storing the T and F values, each value being stored in one memory location. However this approach is very inefficient as it does not utilize the individual bits in a word, which in a typical modern processor is 64 bits long¹. Accordingly, we use the following functions to access individual bits in an array of words.

```
void setBitInArray(unsigned long d[], int numWords, int index){
    // in array d[numWords] set bit index to 1
    int word = index/64;
    int bit = index % 64;
    d[word] = d[word] | (1ULL<<bit);
}
```

¹The NVIDIA GPU, however, has 32 bit words.

```

int tstBitInArray(unsigned long d[], int numWords, int index){
    //    in array d[numWords] return the value of bit index
    int word = index/64;
    int bit = index % 64;
    unsigned long t=(1ULL<<bit);
    return ((d[word] & t)>0)
}

```

With these functions in hand we can express the main loop of the algorithm as follows.

```

for(i=1; i<=numObj; i++){
    int pos = vals[i];
    for(j=0; j<words; j++){
        d[i][j] = d[i-1][j];
        for(j=0; j<=halfSize; j++){
            int thatPos=pos+j;
            if(((tstBitInArray(d[i-1],words,j))== 1) && (thatPos <= halfSize))
                setBitInArray(d[i],words,thatPos);
        }
    }
}

```

This leads to a perfectly viable serial code for the subset-sum problem. Note that the first inner `for` loop simply copies the preceding row to the current row. This could have been done using the test and set bit functions described above, but it is far more efficient to do as a word copy. The second inner `for` loop adds the new cells that are marked T because of the object a_i . For example if cell `d[i-1][j]` is 1 then the element `d[i][j+vals[i]]` in the subsequent row will also be 1. The condition `(thatPos <= halfSize)` ensures that locations outside the table are not erroneously accessed.

2.2 Parallelizing for the Cray XMT

The code presented above is easily parallelized for the Cray XMT by using pragmas. The pragmas are inserted before the inner `for` loops as shown below and ensure

that the chosen loops will be parallelized, even in the event that the compiler suspects dependencies.

```
#pragma mta parallel
for(j=0; j<words; j++)
    d[i][j] = d[i-1][j];
#pragma mta parallel
for(j=0;j<=halfSize;j++){
    ...
}
```

To ensure that only one thread accesses a specific word at a time, we use the XMT's powerful word locking facilities that utilize flag bits associated with *each* 64 bit word. The function `readfe`, 'read full set empty' and `wroteef`, 'write empty set full' allow the user to exercise word level locking. This locking is needed in the function `setBitInArray`, which may be called by multiple threads concurrently.

`readfe` allows the word to be read only if the flag bit of the word is set to full. Upon reading the word, the flag bit is atomically set to empty. `wroteef` allows the word to be written into only when the flag bit is set to empty. As the word is written, the flag bit will atomically be set to full. This process thus ensures that only one thread can access a word at a time.

The `readfe` and `wroteef` instructions block when memory is not ready and a context switch takes place to allow some other thread to start executing. Thus, for a large problem, the CPU is always busy executing some thread or the other.

```
void setBitInArray(unsigned d[], int numWords, int index){
    // in array d[numWords] set bit index to 1
    int word = index/64;
    int bit = index % 64;
    unsigned t=(1ULL<<bit);
    unsigned w=readfe(&d[word]);
    w= w | t;
    wroteef(&d[word],w);
}
```

(a) Performance of Cray XMT

(b) Speedup for problem size 4×10^{11}

Figure 2.1: Parallelizing the subset-sum problem on the Cray XMT using a bit oriented approach

Fig. 2.1(a) plots the time for generating the dynamic programming tables, using the parallelized XMT code. We ran the code using randomly generated numbers in the range of 7500 and 10000 for the number of objects as well as the weights of the objects. This was done for 10 problems each on 1, 2, ..., 128 processors. The plot shows that the code has good scaling and smooth performance. The line for processor 1 is cut short because a single processor was unable to complete all its runs in the dedicated time allocated to us. The observed times show excellent speedup and are mostly within a constant factor of the predicted times, except for the case of 128 processors, as shown in the plot of Fig. 2.2. There are two curves for predicted run times plotted in this figure. These are derived in Appendix A, where we discuss how it is impossible to obtain exact predictions. This is because, on the XMT, instructions with memory accesses take much longer than non-memory access instructions. The exact overlapping of instructions is impossible to predict exactly. We have therefore shown the predicted worst case line, which corresponds to no overlapping, and the predicted best case time which corresponds to complete overlapping. The measured time lies between these two curves indicating that the achieved performance is close to the predicted values.

2.3 Parallelization on Conventional Shared Memory Machines

We cannot just simply achieve parallelization on conventional shared memory machines by using the XMT code. This is because, unlike the XMT, conventional machines do not have the capability to lock individual memory locations. Thus a completely different approach needs to be taken to parallelize the subset-sum algorithm for conventional machines. This is described in Chapter 3. Note, however, that

the XMT code in Section 2.2 is identical to the serial code if the locking operations are omitted, and thus would work correctly on a serial machine.

Chapter 3: Word Oriented Approach

In this chapter we explore ways to parallelize the subset-sum algorithm given in Section 2.1 by using an approach that allows selected words to be concurrently updated. On the Cray XMT this is accomplished by using word level locking. On the other two architectures we use an alternating word approach to achieve this end. We discuss these approaches in the following sections and subsequently evaluate their performance on the three architectures.

3.1 Discussion

Fig. 3.1 shows two consecutive rows from our dynamic programming table. The shaded squares stand for bits that are 1 (i.e., T). When labeling row i from row $i-1$, each T bit in row $i-1$ will set bits that are `size[i]` distant from it. All T bits in row $i-1$ will set bits that are exactly `size[i]` away from them in row i . In other words, the slopes of the arrows between any two rows are equal because they are determined by `size[i]`. In Fig. 3.1 we can see that `word[i-1][j]` can influence the two consecutive words `word[i][j+1]` and `word[i][j+2]`. Fig. 3.2 further shows that a single word can influence *at most* two adjacent words since the distance between any two bits in a word is at most the word length and this can span at most two words in the subsequent row.

It follows that we can split the updating process into two phases. In the first phase, even indexed words in row `i` update the appropriate words of row `i+1`. In the second phase odd indexed words do the same. Since, in any phase, each word can influence only two adjacent words and since the offset is always fixed at `size[i]`, it follows that the the even and odd phases can safely be executed in parallel.

3.2 Alternating Word Code for Shared Memory Machines

Fig. 3.3 presents the alternating word approach described above, as implemented on the IBM x3755. The pragmas are OpenMP (Open Multi-Processing) compiler directives. The `for(start...)` loop carries out the alternation between even and odd words. In general two words of row `i` are modified by ORing them with the source word from row `i-1`, suitably shifted. However if the target bit is the leftmost bit of a word in row `i` then only one word of row `i` is modified. The `if(bit!=0)` statement ensures this. We opted to break up essential lines of the code into smaller operations by using local variables such as `temp11`, etc. This helps the compiler achieve greater flexibility in optimizing and scheduling the code.

3.3 Word Code for Cray XMT

We use a simplification of the alternating word approach in the Cray XMT. Since this machine has full/empty bits which ensure that two threads will not overwrite the same word, we do not need to alternately fill the words. Therefore we skip the outer `for` loop which alternates the words. We also assign temporary variables to improve performance of the code as before, and use `writelf` and `readf` for exclusive access. The resulting code is shown in Fig. 3.4

Figure 3.1: Any bits marked T (shaded) in row $i-1$ will set bits that are $\text{size}[i]$ bits away in row i . The bits in any word in row $i-1$ can influence bits in two words in row i .

Figure 3.2: The T bits in any word in row $i-1$ can influence *at most* two adjacent words in row i .

```

int w;
for(i=1; i<=numObj; i++){
    int size = vals[i];
    int bit, wNew, start;
    UINT upper, temp11, temp12, temp21,temp22;
    #pragma omp parallel for
    for(j=0; j<words; j++)
        d[i][j] = d[i-1][j];
    for(start=0;start<2;start++){
        #pragma omp parallel for private(bit, wNew, upper, temp11, temp12,
                                          temp21, temp22)

        for(w=start;w<words;w=w+2){
            bit = (size % 64);
            wNew = w + (size/64);
            upper=d[i-1][w];
            if(((w+1)*64+size)<=(halfSize+63)){
                temp11=d[i][wNew];
                temp12=temp11|((upper)<<bit);
                d[i][wNew]=temp12;
                if(bit!=0){
                    temp21 = d[i][wNew+1];
                    temp22 = temp21|((upper)>>(64-bit));
                    d[i][wNew+1]=temp22;
                }
            }
        }
    }
}

```

Figure 3.3: Alternating Word Approach implemented using OMP on the IBM x3755

```

int w;
for(i=1; i<=numObj; i++){
    int size = vals[i];
    int mybit, wNew, start;
    #pragma mta block schedule
    #pragma mta assert parallel
    for(j=0; j<words; j++){
        d[i][j] = d[i-1][j];
    }
    #pragma mta block schedule
    #pragma mta assert parallel
    for(w=0; w<words; w++){
        int mybit = (size & 63); //(size%64)
        int wNew = w + (size>>6); //w+(size/64)
        unsigned upper=d[i-1][w];
        unsigned temp11, temp12;
        unsigned temp21, temp22;
        if(((w+1)*64+size)<=(halfSize+63)){
            temp11=readfe(&d[i][wNew]);
            temp12=temp11|((upper)<<mybit);
            writeef(&d[i][wNew],temp12);
            if(mybit!=0){
                temp21 = readfe(&d[i][wNew+1]);
                temp22 = temp21|((upper)>>(64-mybit));
                writeef(&d[i][wNew+1],temp22);
            }
        }
    }
}

```

Figure 3.4: Word Approach implemented on the Cray XMT

This different implementation leads to much better performance on the Cray than by the code used in Section 2, as shown in Fig. 3.5. The improvement in Fig. 3.5 over Fig. 2.1 is due to the far greater efficiency of word operations over bit operations. Note the use of the block schedule pragma above the loop. As discussed in Appendix C, there are several scheduling policies available on the XMT and Block Scheduling gives the best results for our code². The expression for the predicted run times shown in Fig. 3.5(b) are derived in Appendix B. As in Fig. 2.2, the best and worst case predictions are shown. The observed time is close to these predictions, being only slightly greater than the worst case. This demonstrates that in this tightly optimized code, no overlap is taking place and the measured time is quite close to the worst case prediction.

3.4 Alternating Word Approach on NVIDIA FX 5800

We used the NVIDIA FX 5800 because it is a specialized graphical processor unit that can support large numbers of threads. We wrote the alternating word approach code in the NVIDIA programming language CUDA, so that it can be specifically used to run in the GPU. The code can no longer be the main program as then only the host CPU can access it. The code was written as a C program that runs on the host and CUDA kernels that run on the GPU. We opted to write more than one kernel to fulfill our purposes. This ensured that the code worked in exactly the same manner as it previously had been on the IBM x3755 and Cray XMT machines.

²A similar experiment was carried out for the OMP implementation on the IBM x3755, but the result in that case were inconclusive because the performance of that machine deteriorates after 8 processors. We therefore used the default scheduling policy for that machine.

(a) Performance of Cray XMT

(b) Speedup for problem size 4.89×10^{12}

Figure 3.5: Parallelizing the subset-sum problem on the Cray XMT using the word oriented approach

Figure 3.6: Performance of IBM x3755

Figure 3.7: Performance of NVIDIA FX 5800 using two different codes

Figure 3.8: Performance of all three machines

```

__global__ void kernel1(UINT* devPtr, int pitch, int i){
    UINT* row = (UINT*)((char*)devPtr + i * pitch);
    UINT* row2 = (UINT*)((char*)devPtr + (i-1) * pitch);
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    row[tid] = row2[tid];
}

```

Figure 3.9: CUDA Kernel for copying an entire row

For every row in the table we then call `kernel1` to copy the row before it and `kernel2` to perform the alternating word approach of setting the bits as shown in Fig. 3.9 and Fig. 3.10, respectively. Note the variable `pitch` in the argument list of both kernels. When allocating two-dimensional arrays in CUDA it is necessary to use the function `cudaMallocPitch()` which takes the desired row and column sizes as input arguments but allocates a row size called `pitch` which is typically larger than the desired row size and is selected so as to result in better performance. This `pitch` must be used in subsequent accesses to the array.

We call these kernels in the main loop using the following code shown in Fig. 3.11. We must specify the number of blocks and the number of threads per block in `grid_size` and `block_size` respectively, for the kernels to perform their function.

The above implementation gives very good performance as shown in Fig. 3.7. However this code can only be applied to problems that fit within the 4 GB device memory of the FX 5800 card. It is possible to break the dynamic programming matrix into slabs that fit inside device memory and to transfer each slab into host memory as soon as it has been filled. The complete dynamic programming matrix is ultimately reassembled in host memory. This approach allows us to solve a problem of size

```

__global__ void kernel2(UINT* devPtr, int pitch, int width,
                        UINT halfSize, int i, int start){
    int bit, wNew;
    UINT upper, temp11, temp12, temp21, temp22;
    int size = valsDev[i];
    UINT* row = (UINT*)((char*)devPtr + i * pitch);
    UINT* row2 = (UINT*)((char*)devPtr + (i-1) * pitch);
    int w = blockDim.x * blockIdx.x + threadIdx.x;

    if(((w & 1)==start)&&(w<width+1)){
        bit = (size % 32);
        wNew = w + (size/32);
        upper=row2[w];
        if(((w+1)*32+size)<=(halfSize+31)){
            temp11=row[wNew];
            temp12=temp11|((upper)<<bit);
            row[wNew]=temp12;
            if(bit!=0){
                temp21 = row[wNew+1];
                temp22 = temp21|((upper)>>(32-bit));
                row[wNew+1]=temp22;
            }
        }
    }
}

```

Figure 3.10: CUDA Kernel implementing Alternating Word Approach

```

for(i=1; i<rows; i++){
    int start;
    kernel1<<<grid_size,block_size>>>(dDev, pitch, i);
    for(start=0;start<2;start++){
        kernel2<<<grid_size,block_size>>>(dDev, pitch, words+1, halfSize,
                                           i, start);
    }
}

```

Figure 3.11: Calling CUDA Kernels in the Main Program

approximately equal to the host memory. Unfortunately there is a heavy overhead of device-to-host transfers. Nevertheless, we obtain performance that is competitive with the other two machines. The performance of this approach is shown in the plot labeled “Out of core” in Fig. 3.7. The code for this approach is very involved because of the housekeeping required in correctly transferring the slabs from device memory to host memory and is discussed in Appendix D. It is important to note that the out of core curve is much higher than the device memory curve for the same problem sizes. This is caused by the fact that the shared memory code requires the transfer of the entire matrix to the host memory, even if the matrix fits fully within the device memory. Thus for these problem sizes we would prefer to use the device memory code (lower curve). The dashed line in Fig. 3.7 shows the time required to only transfer data from device to host memory. This line is derived from the measured memory bandwidth of 2.67GB/s.

3.5 Comparison of Results

Comparing the plots for IBM x3755, XMT and NVIDIA FX 5800 we note the following:

3.5.1 IBM x3755

This machine scales well to 8 processors, but for 16 processors the time is actually worse than the 4 processor time. The individual curves are initially smooth but start getting jagged and curving upwards at problem size $\approx 3.5 \times 10^{10}$, indicating that this machine cannot handle large amounts of memory accesses. The performance of the machine appears to start degrading significantly at problem size $\approx 4 \times 10^{11}$.

3.5.2 Cray XMT

There is very good scaling in the curves for the Cray XMT up to 16 processors. When we run the program on 64 and 128 processors, we do not achieve the same uniform scaling, in fact scaling is very poor when using a large number of processors for small problem sizes. This is associated with the fact that small problem sizes do not garner much work for 64 to 128 processors and therefore such large processors in combination cannot be used to their maximum potential. However when large problem sizes are present then the 64 and 128 curves appear to be heading towards linear scaling. However larger problem sizes could not be run because of the 1TB memory limitation. If we had more memory, we would be able to accommodate large problem sizes and could have demonstrated good scaling for 64 and 128 processors.

3.5.3 NVIDIA FX 5800

For small problem sizes which can fit inside the device memory of the GPGPU, this machine shows the best performance. Typically device memories for GPGPUs are in the gigabyte range. For larger problems we need to share the host's memory but this can be done only after incurring the heavy overhead of device-to-host transfers. Nevertheless the GPGPU shows good performance equivalent approximately to 3 Opterons.

Chapter 4: Conclusions

The essential feature of our parallel subset-sum algorithm is the alternating word approach which allows the dynamic programming table to be updated without creating race conditions. This approach works very well on conventional shared memory machines as well as GPGPUs. The powerful full/empty word locking facilities of the Cray XMT do not require this alternating word approach.

As we increase the problem size in IBM x3755, the timings start to degrade. Commodity shared-memory machines can only perform well for a limited range of problem sizes because they cannot handle the overhead associated with maintaining cache coherency for large problem sizes [3]. We expect the IBM x3755 curves to deviate even further for larger problems (which we were unable to explore due to the 64GB memory limit on our target hardware).

The Cray XMT is best suited to very large problems. Unlike IBM x3755, it does not perform well at small problem sizes because each individual processor is intrinsically slower. However the processors work well together and, in contrast with IBM x3755, the timings actually improve with increasing problem size. Thus the IBM x3755 performs better than XMT for small problem sizes and the XMT performance is substantially better for large problem sizes.

The performance of the NVIDIA GPGPU when running the out of core algorithm appears to be limited only by the size of the host's memory. We were able to experiment only up to 24GB as the host's connected to our GPGPU had only this much memory. We would however expect no degradation in performance for large memory sizes because no contention for host memory is taking place (unlike the case for conventional shared memory machine, Section 2.3).

In conclusion we can state that the NVIDIA GPGPU is best suited to small problem sizes, the IBM x3755 performs well for medium sizes and the Cray XMT is the clear choice for large problems. For the XMT we expect to see better performance for large problem sizes, should memory larger than 1TB become available. This confirms the hypothesis we set out to demonstrate.

Future research could precede along the following lines:

1. Explore the performance on more powerful GPGPUs with larger device memory.
2. Evaluate the conventional shared memory code on larger configurations, i.e., greater than 16 processors.
3. Combine the GPGPU approach with the conventional shared memory approach.
This would be possible on multicore machines that are connected to 2 or more GPGPUs.
4. Explore the performance of our XMT algorithm on larger configurations, i.e., greater than 128 processors and 1TB memory. The forthcoming XMT-2 which has faster processors and larger memory would certainly demonstrate better performance for very large problems.

Appendix A: Predicted times for the XMT using the Bit Oriented Approach

The annotated output for the XMT code that was obtained from running CANAL (the XMT Compiler Analysis tool) is shown below. Each important part of the code, e.g. loops, which has been assembled has been marked with an number index. We can go to these indices and see how the respective parts of the code have been assembled and what are the number of instructions.

```

        |#pragma mta assert parallel
        |for(i=0; i<=numObj; i++){
        |    #pragma mta assert parallel
        |    for(j=1; j<words; j++){
13 XDD:m|        writexf(&d[i][j],0);
        |    }
        |}
        |

        |//Fill the table
        |for(i=1; i<=numObj; i++){
14 X- |    int pos = vals[i];
        |    #pragma mta assert parallel
        |    for(j=0; j<words; j++){
16 XSD|        d[i][j] = d[i-1][j];
        |    }
        |    #pragma mta assert parallel
        |    for(j=0; j<=halfSize; j++){
17 X-p|        int thatPos=pos+j;
17 XSD|        if(((tstBitInArray(d[i-1],words,j))== 1) &&
        |            ((thatPos) <= halfSize)){
++ function tstBitInArray inlined
```

```

17 XSD|                setBitInArray(d[i],words,thatPos);
++ function setBitInArray inlined
    |      }
    |      }
    |}

```

The following is a summary of the machine language involved in compiling the XMT code. The total number of instructions and the loads and stores are also given for each loop.

```

Loop 13 in main at line 174 in loop 12
  Parallel section of loop from level 3
  Loop summary: 0 loads, 1 stores, 0 floating point operations
                1 instructions, needs 180 streams for full utilization
                pipelined

Loop 16 in main at line 197 in loop 15
  Loop summary: 1 loads, 1 stores, 0 floating point operations
                2 instructions, needs 180 streams for full utilization
                pipelined

Loop 17 in main at line 201 in loop 14
  In parallel phase 5
  Interleave scheduled
  Loop not pipelined: not structurally ok
  Loop summary: 19 instructions, 0 floating point operations
                2 loads, 1 stores, 0 reloads, 0 spills, 3 branches, 0 calls

```

On the XMT, instructions are executed at the rate of 500MIPS but memory load/store instructions only achieve 100MIPS. Loops 13 and 16 are executed `numObj` $\times$ `words` times each, have 1 and 2 memory operations, respectively, and thus take time $(3 \times \text{numObj} \times \text{words}) / 1 \times 10^8 p$ sec, where p is the number of processors.

The total size of the table (which we used in the plots earlier in this Thesis) is `tableSize = numObj` $\times$ `halfSize` (`halfSize` being the length of the bit array). For loop 17, the number of instructions is 19, there are 3 memory operations, and this loop is executed `tableSize` times. It is not possible to compute the exact number

of cycles required by this loop as the precise way the instructions are scheduled is unknown. The most optimistic estimate assumes that all 3 memory operations are completely masked by the 19 instructions leading to a lower bound estimate of $19 \times \text{tableSize}/5 \times 10^8 p$ sec.

The total time for the 3 loops is thus

$$t = (3 \times \text{numObj} \times \text{words})/(1 \times 10^8 p) + (19 \times \text{tableSize})/(5 \times 10^8 p).$$

Since bits are packed into 64-bit words, and $\text{words} = \text{halfSize}/64$, the above expression becomes

$$t = (\frac{3}{64} \times \text{tableSize})/(1 \times 10^8 p) + (19 \times \text{tableSize})/(5 \times 10^8 p)$$

This is one estimate for the best case total time. Another estimate for this is given by using the memory operations in the following expression:

$$t = (\frac{3}{64} \times \text{tableSize})/(1 \times 10^8 p) + (3 \times \text{tableSize})/(1 \times 10^8 p)$$

Thus the best case is determined by the expression

$$t_{\min} = (\frac{3}{64} \times \text{tableSize})/(1 \times 10^8) + \max\{\frac{19}{64} \times \text{tableSize}/(5 \times 10^8 p), \frac{3}{64} \times \text{tableSize}/(1 \times 10^8 p)\}$$

which reduces to

$$t_{\min} = (\frac{3}{64} \times \text{tableSize})/(1 \times 10^8 p) + (3 \times \text{tableSize})/(1 \times 10^8 p)$$

since

$$\frac{3}{64} \times \text{tableSize}/(1 \times 10^8 p) > \frac{19}{64} \times \text{tableSize}/(5 \times 10^8 p)$$

This plot is marked “Predicted Best Case” in Fig. 2.2. The worst case total time is given by the expression:

$$t_{\max} = (\frac{3}{64} \times \text{tablesize}) / (1 \times 10^8 p) + (3 \times \text{tablesize}) / (1 \times 10^8 p) + (16 \times \text{tablesize}) / (5 \times 10^8 p)$$

This is the plot marked “Predicted Worst Case” in Fig. 2.2.

Appendix B: Predicted times for the XMT with Word Approach

In the case of the word-oriented code, we have improved efficiency by only initializing the first row of the table. The time for this is negligible. Note how we use `(size & 63)` instead of `(size%64)` and `(size>>6)` instead of `(size/64)`. This has been done to circumvent a limitation of the XMT's Arithmetic Logic Units.

```

      | //Initialization
1 X   | writexf(&d[0][0],1); //left-most word of first row
      | #pragma mta block schedule
      | #pragma mta assert parallel
      | for(j=1;j<words;j++){ //remaining words of first row
8 XD  |     writexf(&d[0][j],0);
      | }
      | //Fill the table
      | int w;
      | for(i=1; i<=numObj; i++){
9 X-  |     int size = vals[i];
      |     int mybit, wNew, start;
      |     #pragma mta block schedule
      |     #pragma mta assert parallel
      |     for(j=0; j<words; j++){
10 XSD |         d[i][j] = d[i-1][j];
      |     }
      |     #pragma mta block schedule
      |     #pragma mta assert parallel
      |     for(w=0;w<words;w++){
11 X-p |         int mybit = (size & 63); //(size%64)
11 X-p |         int wNew = w + (size>>6); //w+(size/64)
11 X-p |         unsigned upper=d[i-1][w];
      |         unsigned temp11, temp12;
```

```

|         unsigned temp21, temp22;
11 XSD |         if(((w+1)*64+size)<=(halfSize+63)){
11 XSD |             temp11=readfe(&d[i][wNew]);
|             temp12=temp11|((upper)<<mybit);
11 XSD |             writeef(&d[i][wNew],temp12);
11 XSD |             if(mybit!=0){
11 XSD |                 temp21 = readfe(&d[i][wNew+1]);
11 X-p |                 temp22 = temp21|((upper)>>(64-mybit));
11 XSD |                 writeef(&d[i][wNew+1],temp22);
|             }
|         }
|     }
| }

```

The CANAL output is:

```

Loop   8 in main at line 199 in region 7
    In parallel phase 2
    Block scheduled
    Loop summary: 0 loads, 1 stores, 0 floating point operations
                  1 instructions, needs 180 streams for full utilization
                  pipelined

Loop   9 in main at line 204 in region 7

Loop  10 in main at line 209 in loop 9
    In parallel phase 4
    Block scheduled
    Loop summary: 1 loads, 1 stores, 0 floating point operations
                  2 instructions, needs 180 streams for full utilization
                  pipelined

Loop  11 in main at line 214 in loop 9
    In parallel phase 5
    Block scheduled
    Loop not pipelined: not structurally ok
    Loop summary: 14 instructions, 0 floating point operations
                  3 loads, 2 stores, 0 reloads, 0 spills, 3 branches, 0 calls

```

Loops 10 and 11 are executed $\text{numObj} \times \text{words} = \text{tableSize}/64$ times each. Loop 10 has two memory operations and thus takes $(\frac{2}{64} \times \text{tableSize})/(1 \times 10^8)$ sec. Loop 11 has 14 instructions and 5 memory operations, a lower bound on the time required

is determined by the 14 instructions, which yields:

$$t = \frac{14}{64} \times \text{tableSize} / (5 \times 10^8 p) \text{ sec.}$$

An alternative lower bound on the time required is determined by the 5 memory operations, which yields:

$$t = \frac{5}{64} \times \text{tableSize} / (1 \times 10^8 p) \text{ sec.}$$

Thus the best case is determined by the expression

$$t_{\min} = \left(\frac{2}{64} \times \text{tableSize} \right) / (1 \times 10^8) + \max \left\{ \frac{14}{64} \times \text{tableSize} / (5 \times 10^8 p), \frac{5}{64} \times \text{tableSize} / (1 \times 10^8 p) \right\} \text{ sec}$$

which reduces to

$$t_{\min} = \left(\frac{2}{64} \times \text{tableSize} \right) / (1 \times 10^8) + \frac{5}{64} \times \text{tableSize} / (1 \times 10^8 p) \text{ sec}$$

because

$$\frac{14}{64} \times \text{tableSize} / (5 \times 10^8 p) > \frac{5}{64} \times \text{tableSize} / (1 \times 10^8 p)$$

The upper bound is determined by an expression which is non-optimistic, i.e., where the 5 memory operations are not masked by the 14 instructions:

$$t = \frac{5}{64} \times \text{tableSize} / (1 \times 10^8 p) + \frac{9}{64} \times \text{tableSize} / (5 \times 10^8 p) \text{ sec.}$$

Therefore the worst case is determined by

$$t_{\max} = \frac{2}{64} \times \text{tableSize} / (1 \times 10^8) + \frac{5}{64} \times \text{tableSize} / (1 \times 10^8 p) + \frac{9}{64} \times \text{tableSize} / (5 \times 10^8 p) \text{ sec.}$$

The $t_{\min}$ and $t_{\max}$ expressions are plotted in Fig. 3.5(b).

Appendix C: Scheduling Policies on the XMT

We used three directives to experiment the timing of the scheduling in the Cray XMT compiler [7] against that of using no scheduling. To use these directives we just add an additional line on top of the `#pragma mta assert parallel` statement in the code (Section 3.3). This line is `#pragma mta` followed by the specified directive of `block schedule`, `block dynamic schedule` or `interleave schedule`. `block schedule` permits individual threads, assigned to loops, to execute the exact same number of iterations to within 1. `block dynamic schedule` allows each thread to execute a particular block of iterations. A shared counter allows threads to be assigned to blocks on start up and completion. The time taken for the thread to execute particular iterations in the assigned blocks determines how many blocks a thread can execute. `interleave schedule` makes every thread, for the assigned loop, execute the same number of iterations to within 1, but from a range of regularly spaced subsequence of the total iterations. Table C.1 illustrates that `block schedule` has the best performance, with the least time, when used before parallelizing the desired loops.

Table C.1: XMT 128 processor, 5.75×10^{12} bit problem under varying scheduling schemes.

Scheduling Policy	Time
None Specified	244
Block	203
Block Dynamic	326
Interleave	490

Appendix D: Out of Core NVIDIA code

To get around the limitations of limited device memory on the NVIDIA GPU, we use an out of core approach. The full dynamic programming table is divided into chunks of rows. Each chunk is computed on the GPU and then transferred to the host's memory.

To start with, we have a kernel for initializing the first row:

```
--global__ void kernel0(UINT* devPtr){
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    UINT* row = (UINT*)((char*)devPtr);
    if(tid==0)
        row[tid]=1U;
    else
        row[tid]=0U;
}
```

The `kernel1` copies words from one row to the other. `devPtr` points the starting address of the chunk. `devPtr2` is a scratch row used as “glue” between chunks.

```
--global__ void kernel1(UINT* devPtr, UINT* devPtr2, int pitch, int i){
    UINT* row = (UINT*)((char*)devPtr + i * pitch);
    UINT* row2;
    if(i==0)
        row2 = devPtr2;
    else
        row2 = (UINT*)((char*)devPtr + (i-1) * pitch);
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    row[tid] = row2[tid];
}
```

Most of the work is done in `kernel2`:

```

__global__ void kernel2(UINT* devPtr, UINT* devPtr2, int pitch,
                        int width, UINT halfSize, int i, int si, int start){
    int bit, wNew;
    UINT upper, temp11, temp12, temp21,temp22;
    int size = valsDev[si];
    UINT* row = (UINT*)((char*)devPtr + i * pitch);
    UINT* row2;
    if(i==0)
        row2 = devPtr2;
    else
        row2 = (UINT*)((char*)devPtr + (i-1) * pitch);
    int w = blockDim.x * blockIdx.x + threadIdx.x;
    if(((w & 1)==start)&&(w<width+1)){
        bit = (size % 32);
        wNew = w + (size/32);
        upper=row2[w];
        if(((w+1)*32+size)<=(halfSize+31)){
            temp11=row[wNew];
            temp12=temp11|((upper)<<bit);
            row[wNew]=temp12;
            if(bit!=0){
                temp21 = row[wNew+1];
                temp22 = temp21|((upper)>>(32-bit));
                row[wNew+1]=temp22;
            }
        }
    }
}

```

In the main program, the size of the chunk has to be calculated with care. If the problem is too small, it will fit wholly in the device memory of the GPU.

```

int rowsPerChunk =
    MIN((int)ceil(deviceMemory/(sizeof(UINT)*(double)(words+1))),
        (1+numObj));
int chunks = (int)ceil((double)rows/(double)rowsPerChunk);

```

The following part of the main program calls the kernels and copies out the chunks from device memory to host memory as needed.

```

printf("starting main loop...\n"); fflush(stdout);
int thisChunkSize;
gettimeofday(&tvTab1, NULL);
kernel0<<<grid_size,block_size>>>(dDev);    //init first row
for(ch=0;ch<chunks;ch++){                    //for each chunk
    startRow=ch*rowsPerChunk;                  //start row: 0th chunk row
    endRow=MIN(((ch+1)*rowsPerChunk),rows);    //end row: last chunk row + 1
}

```

```

thisChunkSize = endRow - startRow;
for(i=0; i<thisChunkSize; i++){           //iterate through chunk
    if(((ch==0)&&(i==0))){ //don't do this for 0th row of 0th chunk
    }
    else{
        int start;
        int si=ch*rowsPerChunk+i;//the real index
        //eDev is the start of the chunk
        //dDev is the "glue" row
        kernel1<<<grid_size,block_size>>>(dDev, eDev, pitch, i);
        //copy previous row
        for(start=0;start<2;start++){
            kernel2<<<grid_size,block_size>>>(dDev, eDev, pitch, words+1,
                                                halfSize, i, si, start);
        }
    }
}
int lastRowInChunk = rowsPerChunk;
//copy from last row of chunk to "glue" row
cudaMemcpy(eDev, dDev+(lastRowInChunk-1)*(pitch/sizeof(UINT)),
           pitch, cudaMemcpyDeviceToDevice);
int rowsToTransfer = thisChunkSize;
cudaMemcpy2D(d+ch*rowsPerChunk*(words+1), (words+1)*sizeof(UINT),
             dDev, pitch, (words+1)*sizeof(UINT),
             rowsToTransfer,cudaMemcpyDeviceToHost);
//copy this chunk out to host in the appropriate location
}

```

Bibliography

- [1] G. Alverson, R. Alverson, D. Callahan, B. Koblenz, A. Porterfield, and B. Smith. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. In *Proc. Int. Conf. Supercomputing*, pages 188–187, 1992.
- [2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. Int. Conf. Supercomputing*, pages 1–6, 1990.
- [3] Shahid Bokhari and Joel Saltz. Exploring the performance of massively multi-threaded architectures. *Concurrency and Computation: Practice and Experience*, 22(5):588–616, 2010.
- [4] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370 – 1380, 2008.
- [5] Michael R. Garey and David S. Johnson. *Computers and Intractability*. W. H. Freeman, New York, 1979.
- [6] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.
- [7] Cray Inc. *Cray XMT Programming Environment User’s Guide*, 2009. S247914.
- [8] Richard M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.
- [9] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6:40–53, March 2008.
- [10] NVIDIA. *CUDA C Best Practices Guide 3.2*, 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Best_Practices_Guide.pdf.

- [11] NVIDIA. *NVIDIA CUDA Programming Guide 3.0*. 2010. http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf.
- [12] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz. Multi-processor performance on the Tera MTA. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–8, 1998.