Implementing Survey Propagation on Graphics Processing Units

Panagiotis Manolios and Yimin Zhang

College of Computing
Georgia Institute of Technology
Atlanta, GA, 30318
{manolios, ymzhang}@cc.gatech.edu

Abstract. We show how to exploit the raw power of current graphics processing units (GPUs) to obtain implementations of SAT solving algorithms that surpass the performance of CPU-based algorithms. We have developed a GPU-based version of the survey propagation algorithm, an incomplete method capable of solving hard instances of random k-CNF problems close to the critical threshold with millions of propositional variables. Our experimental results show that our GPU-based algorithm attains about a nine-fold improvement over the fastest known CPU-based algorithms running on high-end processors.

1 Introduction

The Boolean satisfiability problem (SAT) has been intensely studied both from a theoretical and a practical point of view for about half a century. The interest in SAT arises, in part, from its wide applicability in domains ranging from hardware and software verification to AI planning. In the last decade several highly successful methods and algorithms have been developed that have yielded surprisingly effective SAT solvers such as Chaff [13], Siege [17], and BerkMin [8]. A major reason for the performance results of recent SAT solvers is that the algorithms and data structures used have been carefully designed to take full advantage of the underlying CPUs and their architecture, including the memory hierarchy and especially the caches [18].

We propose using graphics processing units (GPUs), to tackle the SAT problem. Our motivation stems from the observation that modern GPUs have peak performance numbers that are more than an order of magnitude larger than current CPUs. In addition, these chips are inexpensive commodity items, with the latest generation video cards costing around $500. Therefore, there is great potential for developing a new class of highly efficient GPU-based SAT algorithms. The challenge in doing this is that GPUs are specialized, domain-specific processors that are difficult to program and that were not designed for general-purpose computation.

In this paper, we show how the raw power of GPUs can be harnessed to obtain implementations of survey propagation and related algorithms that exhibit almost an order of magnitude increase over the performance of CPU-based algorithms. We believe that we are the first to develop a competitive SAT algorithm based on GPUs and the first to show that GPU-based algorithms can eclipse the performance of state-of-the-art CPU-based algorithms.

	Pentium EE 840 3.2GHz Dual Core	GeForce GTX7800
FLOPs	25.6 GFLOPs	313 GFLOPs
Memory bandwidth	19.2 GB/sec	54.4 GB/sec
Transistors	230M	302M
Process	90nm	110nm
Clock	3.2GHz	430Mz

Table 1. Performance comparison of NVIDIA's GTX7800 and Intel's Pentium Dual Core EE 840 processor.

The SAT algorithm we consider is survey propagation (SP), a recent algorithm for solving randomly generated k-CNF formulas that can handle hard instances that are too large for any previous method to handle [2, 1]. By "hard" we mean instances whose ratio of clauses to variables is below the critical threshold separating SAT instances from UNSAT instances, but is close enough to the threshold for there to be a preponderance of metastable states. These metastable states make it difficult to find satisfying assignments, and it has been shown in previous work that instances clustered around the critical threshold are hard random k-SAT problems [12, 3, 5].

The rest of the paper is organized as follows. In Section 2, we describe GPUs, including their performance, architecture, and how they are programmed. In section 3, we provide an overview of the survey propagation algorithm. Our GPU-based parallel version of survey propagation is described in Section 4 and is evaluated in Section 5. We discuss issues arising in the development of GPU-based algorithms in Section 6 and conclude in Section 7.

2 Graphical Processing Units

A GPU is a specialized processor that is designed to render complex 3D scenes. GPUs are optimized to perform the kinds of operations needed to support real-time realistic animation, shading, and rendering. The performance of GPUs has grown at a remarkable pace during the last decade. This growth is fueled by the video game industry, a multi-billion dollar per year industry whose revenues exceed the total box office revenues of the movie industry.

The raw power of GPUs currently far exceeds the power of CPUs. For example, Table 1 compares a current Intel Pentium CPU and a current GPU, namely NVIDIA's GTX7800, in terms of floating point operations per second (FLOPs) and memory bandwidth. It is worth noting that the 313 GFLOPs number for the GTX 7800 corresponds to peak GFLOPs available in the GPU's shader, the part of the GPU that is programmable. The total GFLOPs of the GTX 7800 is about 1,300 GFLOPs.

In addition, the performance of GPUs is growing at a faster rate than CPU performance. Whereas CPU speed has been doubling every eighteen months, GPU performance has been doubling about every six months during the last decade, and estimates are that this trend will continue during the next five years. The rapid improvements in

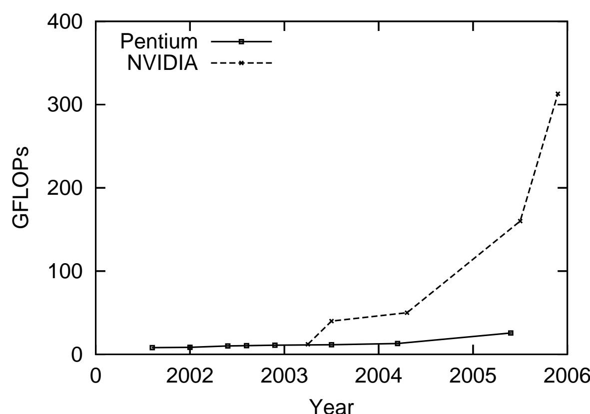

Fig. 1. Comparison of GPUs and CPUs with respect to peak floating point performance. The last two CPU numbers are for dual-core machines. The figure is partly based on data from Ian Buck, at Stanford University.

GPU performance can been seen in Figure 1, which compares the peak floating point performance of GPUs with CPUs over the course of about five years.

The reason why GPUs have such high peak FLOPs and memory bandwidth is that they are architecturally quite different from CPUs; in fact, they should be thought of as parallel stream processors providing both MIMD (Multiple Instruction Multiple Data) and SIMD (Single Instruction Multiple Data) pipelines. For example, NVIDIA's GTX7800 has eight MIMD vertex processes and twenty four SIMD pixel processors. Each of the processors provides vector operations and is capable of executing four arithmetic operations concurrently.

There is increasing interest in using GPUs for general-purpose computation and many successful applications in domains that are not related to graphics have emerged. Examples include matrix computations [7], linear algebra [11], sorting [10], Bioinformatics [15], simulation [6], and so on. In fact, General Purpose computation on GPUs (GPGPU) is emerging as a new research field [9]. Owens et. al. have written a survey paper of the field that provides an overview of GPUs and a comprehensive survey of the general-purpose applications [14].

It is worth pointing out that there are significant challenges in harnessing the power of GPUs for applications. Since GPUs are targeted and optimized for video game development, the programming model is non-standard and requires an expert in computer graphics to understand and make effective use of these chips. For example, GPUs only support 32-bit floating point arithmetic (often not IEEE compliant). They do not provide support for any of the following: 64-bit floating point arithmetic, integers, shifting, and bitwise logical operations. The underlying architectures are largely secret and are rapidly changing. Therefore, developers do not have direct access to the hardware, but must instead use special-purpose languages and libraries to program GPUs. It takes a

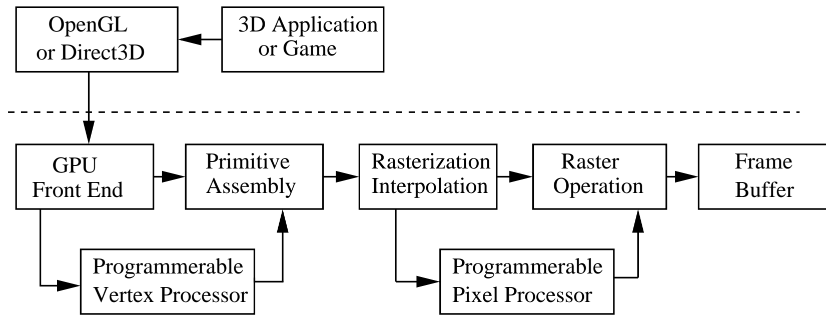

Fig. 2. Graphics pipeline

while for one to learn the many tricks, undocumented features (and bugs), perils, methods of debugging, etc. that are needed for effective GPU programming.

2.1 GPU Pipeline

Nearly all current GPUs use a graphics pipeline consisting of several stages, as outlined in Figure 2. The graphics pipeline takes advantage of the inherent parallelism found in the kinds of computations used in 3D graphics.

An application, say a game, communicates with a GPU through a 3D API such as OpenGL or Direct3D. The application uses the API to send commands and the vertex data modeling of the object to be rendered to the GPU. Objects are modeled by a sequence of triangles, each of which is a 3-tuple of vertices. After receiving this stream of vertices, the GPU processes each vertex using a vertex shader, an application-specific program that runs on the vertex processor and which computes vertex-specific values, such as position, color, normal vector, etc. These transformed vertices are then assembled into triangles (by the primitive assembly stage) and the vertex information is then used to perform interpolation and rasterization, producing a 2D raster image. Next, a pixel shader, an application-specific program running on the pixel processor is used to compute the color values of pixels. The pixel shader can access information stored on *textures*, memories organized as cubes. (In GPGPU applications, texture memory is used in lieu of main memory.) Finally, a color vector containing the four values R(red), G(green), B(blue), and A(alpha) is output to the frame buffer and displayed on the screen.

Notice that there are two kinds of programmable processors in the graphics pipeline, the vertex processor and pixel processor (also called the fragment processor). Both types of processors are capable of vector processing and can read from textures. However, the vertex processors are MIMD processors, whereas the pixel processors are SIMD processors.

One major difference between GPUs and CPUs is that GPUs are capable of "gathering" but not "scattering." Roughly speaking, gathering means being able to read any part of memory, while scattering means being able to write to any part of memory. The

memories we are referring to are textures. Both vertex shaders and pixel shaders are capable of gathering, but as can be seen in Figure 2 vertex shaders cannot directly output data. Instead, they output a fixed number of values to the next stage in the pipeline. In contrast, pixel shaders are able to output data, by writing into the frame buffer. The output for each pixel is a 4-word value representing a color vector. Pixel shaders can also write to textures by utilizing OpenGL extensions. A major limiting factor for GPGPU applications is that the amount of information and its location are fixed before the pixel is processed. In graphics applications, this is not much of a limitation because one typically knows where and what to draw first.

For general purpose computation, pixel processors are usually preferable to vertex processors. There are several reasons for this. First, GPUs contain more pixel processors than vertex processors. Second, pixel processors can write to texture memory, whereas vertex processors cannot. Finally, pixel shader texturing is more highly optimized (and thus much faster) than vertex shader texturing.

2.2 OpenGL

Recall that the architectures of GPUs are closely guarded secrets. Therefore, developers do not have direct access to GPUs and instead have to access the chips via a software interface. One popular choice, which is what we use in this paper, is OpenGL (Open Graphics Library), a specification for a low-level, cross-platform, cross-language API for writing 3D computer graphics applications.

Currently, the OpenGL specification is managed by ARB, the OpenGL Architecture Review Board, which includes companies such as NVIDIA, ATI, Intel, HP, Apple, IBM, etc. OpenGL is an industry standard that is independent of the operating system and underlying hardware. Microsoft has its own API, DirectX, which is dedicated to the Window operating system.

OpenGL is very popular for general purpose computing with GPUs, in part due to its ability to quickly extend the specification with extensions in response to GPU hardware developments. These extension enable developers to more fully take advantage of the new functionality appearing in graphics chips. One example of this is the Frame Buffer Object (FBO), an essential component for general purpose computing with GPUs. The FBO allows shader programs to write to a specified texture, instead of writing to the frame buffer. This is quite useful because in the graphics pipeline, no matter what value is written to the frame buffer, it is turned into a value in the interval [0..1], which makes writing non-graphics applications quite difficult. A further benefit of using an FBO is that we can write to a texture and then use this texture as input in the next pass of the rendering process.

2.3 The Cg Programming Language

Cg (C for Graphics) is a high-level language for programming vertex and pixel shaders, developed by NVIDIA. Cg is based on C and has essentially the same syntax. However, Cg contains several features that make it suitable for programming graphics chips. Cg supports most of the operators in C, such as the Boolean operators, the arithmetic operators, etc., but also includes supports for vector data types and operations. For example,

```
float4 main(uniform samplerRECT exampletexture, float4 pos : WPOS) {
         float4 color;
         color = texRECT(exampletexture, pos.xy);
         return color;
}
```

Fig. 3. This is an example of a pixel shader using Cg.

```
void draw() {
        cgGLBindProgram(UpdateEta);
        glDrawBuffer(GL_COLOR_ATTACHMENT3_EXT);
        cgGLSetTextureParameter(etavarParam, varTex);
        cgGLEnableTextureParameter(etavarParam);
        glBegin(GL_QUADS);
           glVertex2f(0.0, 0.0);
           glVertex2f(100, 0.0);
           glVertex2f(100, 100);
           glVertex2f(0.0, 100);
        glEnd();
}
```

Fig. 4. This is an OpenGL code snippet.

it supports float4, a vector of four floating point numbers, and it supports MAD, a vector multiply and add operator. Cg also supports several other graphic-based operations, *e.g.*, it provides functions to access the texture, as shown in the Cg example in Figure 3.

In addition to the features appearing in Cg that do not appear in C, there are also limitations in Cg that do not appear in C. For example, while user-defined functions are supported, recursion is not allowed. Arrays are supported, but array indices must be compile-time constants. Pointers are not supported; however, by using texture memory, which is 2-dimensional, they can be simulated by storing the 16 high-level bits and the 16 low-level bits in the in the x and y coordinates, respectively. Loops are allowed only when the number of loop iterations is fixed. In addition, the `switch`, `continue`, and `break` statements of C are not supported.

Figure 3 gives an example of a Cg program. The main entry of a Cg program can have any name. In the above example, we have a function `main` that is a pixel shader whose output is a float4 representing a 4-channel color vector. Our simple pixel shader takes a single texture, `exampletexture`, and a single float4, `pos`, as input and samples a color value (using the Cg function `texRECT`) from position `pos` in the texture.

In Figure 4, we provide an OpenGL example that simply draws a rectangle of size 100x100 pixels on the screen. It first installs the pixel shader program, `UpdateEta` (from Figure 3); then it chooses the texture to write to, `GL_COLOR_ATTACHMENT3_EXT`; then it selects the texture to read from, `varTex`; and finally it sends the rendering command (starting at `glBegin(GL_QUADS)`). Executing the rendering command results in running 10,000 pixel shader programs, one per pixel in the 100x100 area. Each pixel

shader program outputs a color, as described above. Notice, that to utilize the GPU, we have to “draw” something, and this means that the position of the output has to be fixed.

3 Survey Propagation

Survey Propagation is a relatively new incomplete method based on ideas from statistical physics and spin glass theory in particular [1]. SP is remarkably powerful, able to solve very hard k-CNF problems, *e.g.*, it can solve 3-CNF problems near threshold with over 10^7 propositional variables.

In this section, we provide a brief overview of the algorithm. We start by recalling that a factor graph can be used to represent a SAT problem. It is bipartite graph whose nodes are the propositional variables and clauses appearing in the SAT instance. There is an edge between a variable and a clause iff the variable appears in the clause. If the clause contains a positive occurrence of variable, the edge is drawn with solid line; otherwise, it is drawn with a dotted line. An example is shown in Figure 5.

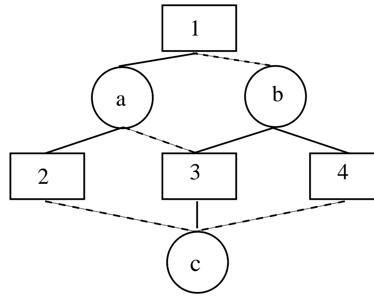

Fig. 5. The factor graph for $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee \neg x_4)$.

Let C, V represent sets of clauses and variables, respectively. We will use $a, b, c, \ldots$ to denote clauses and $i, j, k, \ldots$ to denote variables. We define $V(a)$ to be the set of variables appearing in clause a and similarly $C(i)$ denotes the set of clauses containing variable i. $C^+(i)$ is the subset of $C(i)$ containing the clauses in which i appears positively; and $C^-(i)$ is the subset of $C(i)$ containing clauses in which i appears negatively (negated).

$$
\begin{array}{llll}
\text{(3.1)} & \text{if } a \in C^+(i) & C^u_a(i) = C^-(i); & C^s_a(i) = C^+(i) \setminus \{a\} \\
& \text{if } a \in C^-(i) & C^u_a(i) = C^+(i); & C^s_a(i) = C^-(i) \setminus \{a\}
\end{array}
$$

The SP algorithm is an iterative message-passing algorithm that for every edge $\langle a, i \rangle$ in the factor graph passes messages consisting of a floating point number, $\eta_{a \to i}$, from clause a to variable i and passes messages consisting a 3-tuple of floating point numbers, $\Pi_{i \to a} = \langle \Pi^u_{i \to a}, \Pi^s_{i \to a}, \Pi^0_{i \to a} \rangle$, from variable i to clause a. This process is initialized by randomly assigning values to $\eta_{a \to i}$ from the interval $(0..1)$ for all edges $\langle a, i \rangle$

in the factor graph. The process is then repeated, where each iteration is called a *cycle*, as described below. As is discussed in [1], the messages can be thought of corresponding to probabilities of warnings. The value $\eta_{a\to i}$, sent from a to i, corresponds to the probability that clause a sends a warning to variable i, which it will do if it receives a "u" symbol from all of its other variables. In the other direction, the triple $\Pi_{i\to a} = \langle \Pi^u_{i\to a}, \Pi^s_{i\to a}, \Pi^0_{i\to a}\rangle$ sent from i to a corresponds to the probability that i sends the "u" symbol (indicating that it cannot satisfy a) or sends the "s" symbol (indicating that it can satisfy a) or sends the "0" symbol (indicating that it is indifferent). The formal definitions follow.

$$\eta_{a\to i} = \prod_{j\in V(a)\setminus\{i\}} \left[\frac{\Pi^u_{j\to a}}{\Pi^u_{j\to a}+\Pi^s_{j\to a}+\Pi^0_{j\to a}}\right] \tag{3.2}$$

$$\prod^u_{j\to a} = \left[1-\prod_{b\in C^u_a(j)}(1-\eta_{b\to j})\right]\prod_{b\in C^s_a(j)}(1-\eta_{b\to j}) \tag{3.3}$$

$$\prod^s_{j\to a} = \left[1-\prod_{b\in C^s_a(j)}(1-\eta_{b\to j})\right]\prod_{b\in C^u_a(j)}(1-\eta_{b\to j}) \tag{3.4}$$

$$\prod^0_{j\to a} = \prod_{b\in C(j)\setminus\{a\}}(1-\eta_{b\to j}) \tag{3.5}$$

The message passing described above is iterated until we attain convergence, which occurs when each of the η values changes less than some predetermined value. Such a sequence of cycles is called a *round*. At the end of a round, we identify a predetermined fraction[1] of variables that the above process has identified as having the largest bias and assign them their preferred values. Having fixed the values of the variables just identified, we perform Boolean Constraint Propagation (BCP) to reduce current SAT problem to simpler one. If the ratio of clauses to variables becomes small, then the problem is under-constrained and we can use Walk-SAT or some other SAT algorithm to quickly find a solution. Otherwise, we again apply the SP algorithm to the reduced SAT problem. Of course, it is possible that either BCP encounters a contradiction or that SP fails to converge, in which case the algorithm fails.[2]

Most of running time of SP is spent trying to converge. Notice that this part of the algorithm requires performing a large number of memory reads and writes and also requires a large number of floating point operations. This is exactly what GPUs excel at doing, which is why we have chosen to develop a GPU-based SP algorithm.

[1] We use 1 percent, the same percentage used in the publicly available implementation by the authors of survey propagation.

[2] In our implementation, we say that SP fails to converge if it takes more that 1,000 cycles, which is the same parameter used in the code by the authors of survey propagation.

4 Parallel SP on GPU

The basic idea for how to parallelize the SP algorithm is rather straightforward, because the order in which edges in the factor graph are updated does not matter. Therefore, we can implement the SP algorithm by running a program per edge in the factor graph, whose sole purpose is to update the messages on that edge. We can then update the messages concurrently. That is the basic idea of the GPU algorithm. In more detail, we ask the GPU to "draw" a quad on the screen where there is one pixel per edge. This allows us to use a pixel shader per edge to compute and update the edge messages, and to store the result in the texture memory. Of course, the CPU and GPU have to communicate after every round of the SP algorithm, so that the GPU can inform the CPU of what variables to fix, so the CPU can perform the BCP pass, and so the CPU can then update the GPU's data structures to reflect the implied literals. The CPU also determines when the clause to variable ratio is such that Walk-SAT should be used.

Given the irregular architecture of GPUs and the difficulty in programming them, one must carefully consider the data structures used and the details of the algorithm, something we now do.

4.1 Data Structures

When using GPUs, the textures are the only place where we can store large amounts of data. Therefore, all of the data used by our algorithm is encoded into textures, which are rectangular areas of memory. One read-only texture is used to represent the factor graph. In it we store, for each clause, pointers to all the literals appearing in that clause (*i.e.*, pointers to all the edges in the factor graph). The pointers for a particular clause are layed out sequentially, which make it easy for us to traverse the edges of a given clause.

We also have three read-write textures, which are used to store information about the variables, clauses, and edges. The variable texture has an entry per variable; similarly the clause and edge textures have entries per clause and edge, respectively.

The main components of the variable texture for entry j include $\prod_{b\in C^+(j)}(1-\eta_{b\to j})$, $\prod_{b\in C^-(j)}(1-\eta_{b\to j})$, and a pointer to the edge texture. In the edge texture, the edges with the same variable are layed out sequentially, so the pointer is to the first such edge and we also store the number of edges containing variable j.

The main components of the clause texture for entry a include a pointer into the read-only texture (which points to the first pointer in the read-only texture for that clause) and the value $\prod_{j\in V(a)}\left[\frac{\Pi^u_{j\to a}}{\Pi^u_{j\to a}+\Pi^s_{j\to a}+\Pi^0_{j\to a}}\right]$.

The main components of the edge texture for entry $\langle a,i\rangle$ are a pointer to the variable i (in the variable texture), a pointer to the clause a (in the clause texture), and $\eta_{a\to i} = \prod_{j\in V(a)\setminus\{i\}}\left[\frac{\Pi^u_{j\to a}}{\Pi^u_{j\to a}+\Pi^s_{j\to a}+\Pi^0_{j\to a}}\right]$.

In current version of OpenGL, the maximum texture size is limited to 256MB, and this is a major restriction because it limits the size of the problems we can consider. We note that the amount of memory available on GPUs is constantly increasing (already GPUs with 1GB memory are available) and that it is possible to use multiple GPUs

together. Also, the OpenGL size constraints on textures will eventually be relaxed, but for now, one can distribute the data across multiple textures.

4.2 Algorithm

The algorithm is given as input a factor graph encoded in the textures as described above. Also, the η values are initialized with randomly generated numbers from the interval $(0..1)$. If successful, the algorithm returns a satisfying assignment. As previously described, the algorithm consists of a sequence of rounds, the purpose of which is to converge on the η values. A single round of the algorithm consists of a sequence of cycles, each of which includes four GPU passes, where we assume that we start with the correct η values and show how to compute the η values for the next cycle. Recall that computing on a GPU means we have to draw quads using OpenGL, which in turn means that the computation is being performed by pixel shaders. We omit many of the details and focus on the main ideas below.

1. For each variable j, compute $\prod_{b \in C^+(j)}(1-\eta_{b\to j})$ and $\prod_{b \in C^-(j)}(1-\eta_{b\to j})$ by iterating over all of the edges that variable appears in. Recall that we have a pointer to the first such edge in the variable texture and that we know what the number of such edges is.
2. For each clause a, compute $\prod_{j \in V(a)}\left[\frac{\Pi^u_{j\to a}}{\Pi^u_{j\to a}+\Pi^s_{j\to a}+\Pi^0_{j\to a}}\right]$ by iterating over all the edges this clause appears in. Recall that that we have a pointer to the read-only texture and we know the number of such edges. The pointer to the read-only memory points to the first such edge in the edge texture and the next pointer points to the next edge, and so on. By iterating and following the variable pointers in the edge texture, we can compute the above value. This is because we can use the values stored in the variable texture to compute $\prod^u_{j\to a}$, $\prod^s_{j\to a}$, and $\prod^0_{j\to a}$ for each variable j occurring in a.
3. For each edge $\langle a,i\rangle$, compute $\eta_{a\to i} = \prod_{j \in V(a)\setminus\{i\}}\left[\frac{\Pi^u_{j\to a}}{\Pi^u_{j\to a}+\Pi^s_{j\to a}+\Pi^0_{j\to a}}\right]$. This can be done by iterating over the elements in the edge texture and using the pointers to the variable and clause of the edge. All that is required is a simple division, given the information already stored in the textures (and after recomputing $\prod^u_{j\to a}$, $\prod^s_{j\to a}$, and $\prod^0_{j\to a}$).
4. Use an occlusion query to test for convergence. If so, this round is over and the GPU and CPU communicate as described previously. Otherwise, goto step 1. An occlusion query is a way for the GPU to determine how many of the pixel shaders have updated the frame buffer. In our case, if the difference between consecutive η values is below a certain threshold, the pixel shader does not update the frame buffer. If the occlusion query returns 0, that means that all of the pixel shaders were killed, *i.e.*, the algorithm has converged.

We note that a GPU's inherent limitations with respect to the support of dynamic data structures can lead to inefficiencies. For example, after BCP, the length of clause may be reduced. Unfortunately, due to the restrictions imposed by Cg, GPU-based programs cannot take advantage of reduced clause sizes and will still have to scan for k

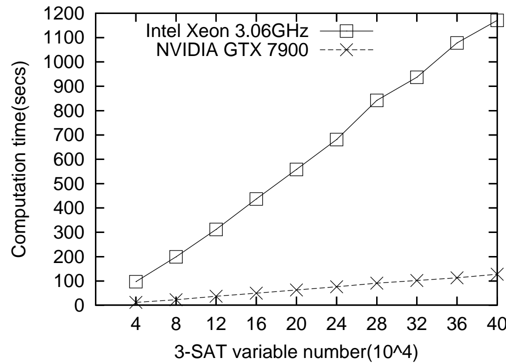

Fig. 6. A comparison between our GPU-based algorithm and the fastest CPU-based algorithm for survey propagation on random 3-SAT instances, with a clause to variable ratio of 4.2.

literals. Fortunately, if we lay out the literals in a clause in a sequential fashion (which we do), then there is a negligible effect on performance.

5 Experimental Results

We implemented our GPU-based survey propagation algorithm using Cg1.4, C++, and OpenGL2.0. The experiments were run on an AMD 3800+ 2.4GHz machine with an NVIDIA GTX 7900 GPU. The operating system we use is 32-bit WindowsXP. We note that this is a 64-bit machine and we expect to get better performance numbers if we use it for 64-bit computation, but since the NVIDIA GTX 7900 is a rather new GPU, the only drivers we could find were for 32-bit Windows. We also note that using NVIDIA's SLI (Scalable Link Interface) technology, we can use two NVIDIA GPUs, which should essentially double our performance numbers.

The CPU-based survey propagation program we used is from the authors of the survey propagation algorithm [1] and is the fastest implementation of the algorithm we know of. We ran the survey propagation algorithm on the fastest machine we had access to, which is an Intel(R) Xeon(TM) CPU 3.06GHz with 512 KB of cache, running Linux Redhat. (We did not use the same machine we ran the GPU experiments on because the Intel machine is faster.) The experimental data we used is available upon request.

In Figure 6, we compare the two algorithms on a range of 3-SAT instances, where the clause to variable ration is 4.2; this means that the problems are hard as they are close to the threshold separating satisfiable problems from unsatisfiable problems [12]. The number of variables ranges from 40,000 to 400,000 and each data point corresponds to the average running time for three problems of that size. As is evident in Figure 6, our algorithm is over nine times as fast as the CPU based algorithm.

In Figure 7, we compare the two algorithms on a range of hard 4-SAT instances, where the clause to variable ration is 9.5. The results for the 4-SAT instances are similar to the results we obtained in the 3-SAT case. That is, for hard 4-SAT instances, our

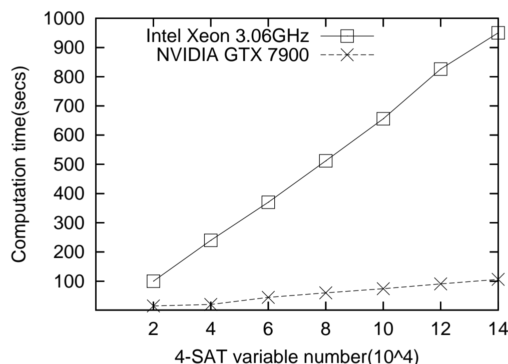

Fig. 7. A comparison between our GPU-based algorithm and the fastest CPU-based algorithm for survey propagation on random 4-SAT instances, with a clause to variable ratio of 9.5.

GPU-based algorithm attains about a nine-fold improvement in running times over the best known CPU-based algorithm.

6 Observations on Programming With GPUs

In this section, we outline some observations about programming with GPUs that we think are relevant to the SAT community. The currently available information on GPU programming is mostly geared to the graphics community, and our experience has been that it takes a while for non-specialists to understand. Hopefully, our observations will help to speed up the process for researchers interested in applying GPUs to SAT and similar problems. A good source of information on this topic is the GPGPU Website [9].

When considering using GPUs for general purpose computing, it is important to choose or develop a parallel algorithm. Recall that these processors are best thought of as parallel stream processors and all algorithms that have been successfully implemented on these chips are parallel algorithms. In fact, GPUs are a poor choice for performing reductions, *e.g.*, selecting the biggest element in an integer array turns out to be very difficult to implement efficiently on a GPU.

It is also important to be aware GPUs do not currently support integer operations. You may have noticed that Cg does have integer operations, but these are compiled away and are in fact emulated by floating point operations. Another important difference between CPU and graphics processors is that GPUs do not perform well in the presence of branch instructions, as they do not support branch prediction. Also, reading data from the GPU to the CPU is often a bottleneck. Finally, a major limitation of GPUs is that the per pixel output is restricted to be a four-word vector (extensions allowing sixteen four-word vectors are also currently available), which effectively rules out the use of GPUs for algorithms that do not fit this framework.

Since many optimization algorithms are iterative in nature, they may well be good candidates for implementing on graphics processors. When doing this, we suggest that

one carefully encodes the problem into the texture. It is important to do this in a way that attains as much locality as possible because GPUs have very small caches, which means that it is crucial to read memory sequentially, as random access to memory will have a detrimental effect on performance.

It is also often necessary to divide algorithms into several passes. For example, recall that each pixel shader only outputs one four-word vector; if more than four words are needed, then multiple passes have to be used. The general idea is to partition algorithms into several steps, each of which performs a specific function and saves intermediate results to a texture. Subsequent passes can then use the result of the previous passes.

One optimization that is quite useful is to test convergence by using an occlusion query. Without this, one has to use the CPU to test for convergence, which will greatly affect performance. In contrast, an occlusion query gives precise information and can be pipelined, so it has negligible impact on the performance of GPUs.

7 Conclusions and Future work

In this paper, we have shown how to harness the raw power of GPUs to obtain an implementation of survey propagation, an incomplete method capable of solving hard instances of random k-CNF problems close to the critical threshold. Our algorithm exhibits about an order of magnitude increase over the performance of the fastest CPU-based algorithms. As far as we know, we are the first to develop a competitive SAT algorithm based on graphics processors and the first to show that GPU-based algorithms can eclipse the performance of state-of-the-art CPU-based algorithms running on high-end processors.

We foresee many opportunities to exploit the power of GPUs in the context of SAT solving and verification algorithms in general. Graphics processors are undergoing rapid development and will almost certainly incorporate many new features that make them even more suitable for general purpose computation in a few years. Consider that programmable GPUs were first introduced in 2002 and now they support a rich instruction set and surpass the most powerful currently available CPUs both in terms of memory bandwidth and peak floating point performance.

For future work, we plan to add further improvements to our algorithm and want to explore using GPUs to help speed up complete SAT algorithms such as those based on DPLL [4]. One simple idea is to use GPUs as coprocessors which are used to compute better heuristics that the DPLL algorithm running on the CPU can take advantage of. Another idea we are exploring is the use of other non-standard processors such as the Cell processor.

References

1. A. Braunstein, M. Mezard, and R. Zecchina. Survey propagation: an algorithm for satisfiability. *Random Structures and Algorithms*, 27:201–226, 2005.
2. A. Braunstein and R. Zecchina. Survey and belief propagation on random k-SAT. In *6th International Conference on Theory and Applications of Satisfiability Testing, Santa Margherita Ligure, Italy (2003)*, volume 2919, pages 519–528, 2003.

3. S. A. Cook and D. G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In Du, Gu, and Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35, pages 1–17. American Mathematical Society, 1997.
4. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
5. O. Dubois, R. Monasson, B. Selman, and R. Zecchina. Statistical mechanics methods and phase transitions in optimization problems. *Theoretical Computer Science*, 265(3–67), 2001.
6. Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover. GPU cluster for high performance computing. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, pages 47–47, 2004.
7. K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–138, 2004.
8. E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT-solver. In *Design, Automation, and Test in Europe (DATE '02)*, pages 142–149, Mar 2002.
9. GPGPU. General-Purpose Computation using GPUs, 2006. `http://www.gpgpu.org`.
10. P. Kipfer and R. Westermann. Improved GPU sorting. In Pharr and Fernando [16], pages 733–746.
11. J. Kruger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.
12. M. Mezard and R. Zecchina. The random k-satisfiability problem: from an analytic solution to an efficient algorithm. *Physical Review E*, 66:056126, 2002.
13. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC'01)*, pages 530–535, 2001.
14. J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, Aaron, E.Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, 2005.
15. B. R. Payne. *Accelerating Scientific Computation in Bioinformatics by Using Graphics Processing Units as Parallel Vector Processors*. PhD thesis, Georgia State University, Nov. 2004.
16. M. Pharr and R. Fernando, editors. Addison Wesley, Mar 2005.
17. L. Ryan. Siege homepage. See URL `http://www.cs.sfu.ca/` ~`loryan/personal`.
18. L. Zhang and S. Malik. Cache performance of SAT solvers: A case study for efficient implementation of algorithms. In S. M. Ligure, editor, *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003)*, pages 287–298, 2003.