A Fast Implementation of the Octagon Abstract Domain on Graphics Hardware

Francesco Banterle and Roberto Giacobazzi

Dipartimento di Informatica
Università degli Studi di Verona
Strada Le Grazie 15, 37134 Verona, Italy
E-mail: frabante@gmail.com and roberto.giacobazzi@univr.it

Abstract. We propose an efficient implementation of the Octagon Abstract Domain (OAD) on Graphics Processing Unit (GPU) by exploiting stream processing to speed-up OAD computations. OAD is a relational numerical abstract domain which approximates invariants as conjunctions of constraints of the form $\pm x \pm y <= c$, where x and y are program variables and c is a constant which can be an integer, rational or real. Since OAD computations are based on matrices, and basic matrix operators, they can be mapped easily on Graphics Hardware using texture and pixel shader in the form of a kernel that implements matrix operators. The main advantage of our implementation is that we can achieve sensible speed up by using a single GPU, for each OAD operation. This can be the basis for an efficient abstract program analyzer based on a mixed CPU-GPU architecture.
Keywords: Octagon Abstract Domain, General Processing on GPU, Parallel Computing, Abstract Interpretation, Static Program Analysis.

1 Introduction

The study of stream processing (computing and programming) is recently gaining interest as an alternative and efficient methodology for allowing parallel processing in many fields of computer science. The paradigm is essentially based on defining a set of compute-intensive operations (called kernels) which are applied to each element in the stream. The growing success of this technology is related with the impressive grow in computational power of dedicated stream processing units (e.g., for graphic processing and more in general for digital signal processing) and in their relatively cheap costs. Recently researchers have become interested in developing algorithms for GPUs. These algorithms were at the beginning designed only for Computer Graphics purposes, but the high computational power offered pushed researchers to explore the possibilities of using GPU in more general tasks, leading to the so called General Purpose Computing on GPU (GP-GPU). GPUs were applied with success in various fields: Database [10], numerical methods [2, 14, 9], and scientific computing [11, 8]; see [21, 23] for excellent surveys on GP-GPU.
In this paper we propose a new programming methodology to handle massive

computations in numerical (relational) abstract domains. We consider stream processors and programming as an efficient and fast methodology for implementing the basic operators of abstract domains involving large matrices and massive data sets. Among the wide spectrum of numerical abstract domains, octagons [18] plays a key role due to their relational structure and affordable computational costs. Octagons provide a way to represent a system of simplified inequalities on the sum and difference of variable pairs, i.e. they represent constraints of the form $\pm x \pm y <= c$, where x and y are program variables and c is a constant which can be an integer, rational or a real number automatically inferred. Their typical implementation is based on Difference Bound Matrices (DBM), a data structure used to represent constraints on differences of pairs of variables. Efficiency is a key aspect in this implementation: The space is constrained for n-variables in $O(n^2)$ and time is constrained in up to $O(n^3)$. This makes the relational analysis based on octagons applicable to large scale programs, e.g., those considered in the ASTRÉE static analyzer [5] which employs programs having more than 10.000 global variables, most of them floating-point, in long-time iterations (about 3.6×10^6 iterations of a single loop). We prove that an important speed-up factor can be obtained by handling DBM in a stream-like computation model. In particular we exploit the structure of Graphics Processing Unit or (GPU), also called Visual Processing Unit, for an efficient and fast implementation of the abstract domain of octagons, in particular for a fast implementation of the basic operations on DBM. GPUs provide a dedicated hardware architecture for graphics rendering by exploiting a highly parallel structure making graphic computations far more effective than typical CPUs for a wide range of complex algorithms. This architecture is particularly suitable for operations on matrices, and therefore for handling operations on DBM. A typical GPU implements a number of graphics primitive operations in a way that implements stream processing: First, the data is gathered into a stream from memory. The data is then operated upon by one or more kernels, where each kernel comprises several operations. Finally, the live data is scattered back to memory. The static analyzer designed in our implementation is based on a CPU which manages the control flow and the GPU which performs the basic operators on the domain. Octagons are represented as 2D textures and the basic operations on octagons are implemented as kernel operations on their fragments. These operations can be performed in parallel on the texture due to their independence and thanks to the high degree of parallelism of the SIMD architecture like GPUs. The pipeline of the analyzer is therefore as follows: each time the analyser reaches a program point, the corresponding instruction is decomposed into basic abstract operations on octagons. The GPU is then activated to perform the basic computations, leaving the result in the video memory. As a result of our implementation, we obtain a sensible speed-up of several orders for simple operators on octagons (i.e., intersection, union, assignment, test guard, widening), and a speed-up around 24 times for the basic operation of octagon closure, which is performed each time octagons have to be merged. The bottleneck in our system is given by the test

guard operator. This is due to the SIMD architecture of GPU which requires the spreading of the computation along the whole texture.

2 The Octagon Abstract Domain

The octagon abstract domain introduced by Miné in [18] is a (weakly) relational abstract domain which provides an upper approximation of program invariants as conjunctions of constraints of the form $\pm x \pm y <= c$, where x and y are program variables and c is a constant which can be an integer, rational or a real. This abstract domain fits between the less precise linear-time non-relational abstract domain of intervals and the exponential-time relational approximation of convex polyhedra. Given a set of program variables $\mathcal{V} = \{V_1, \ldots, V_n\}$, we consider a set of enhanced variables: $\mathcal{V}' = \{V'_1, \ldots, V'_{2n}\}$ where for any $V_i \in \mathcal{V}$ we have both a positive form V'_{2i-1}, denoted V_i^+, and a negative form V'_{2i}, denoted V_i^-, in $\mathcal{V}'$. A Difference Bound Matrix, or DBM for short, $\mathbf{m}$ is a $n \times n$ square matrix with elements in a field $\mathbb{Z}$ or $\mathbb{R}$ [18]. The element at line i and column j, denoted $\mathbf{m}_{ij}$ equals a constant c if there is a constraint of the form $V_j - V_i \leq c$, and $+\infty$ otherwise. Thus a conjunction of octagonal constraints in $\mathcal{V}$ can be represented as a DBM with $2n$ dimension. In particular, a Galois connection has been established between DBM and sets of tuples of values:

$$\gamma(\mathbf{m}) = \{\ \langle v_1, \ldots, v_{2n}\rangle \,|\, \forall i, j \leq 2n.\ v_j - v_i \leq \mathbf{m}_{ij}\ \} \cap \{\ \langle v_1, \ldots, v_{2n}\rangle \,|\, v_{2i-1} = -v_{2i}\ \}$$

is the octagon represented by the $2n$ dimension DBM $\mathbf{m}$. This $2n$ space is isomorphic to a n-dimensional space which represent a convex structure having an octagon-like shape.
The set of (coherent) DBM, denoted **cDBM**, enriched with a bottom (empty) element $\bot^{\mathbf{cDBM}}$ representing the empty set $\varnothing$ and a top element $\top^{\mathbf{cDBM}}$ representing the whole space, i.e., such that $\forall i, j.\ \top_{ij}^{\mathbf{cDBM}} = +\infty$, and ordered w.r.t. set inclusion, i.e., $\mathbf{m} \sqsubseteq \mathbf{n}$ iff $\forall i, j :\ \mathbf{m}_{ij} \leq \mathbf{n}_{ij}$, forms a complete lattice, where a DBM is coherent if $\forall i, j.\ \mathbf{m}_{i,j} = \mathbf{m}_{\bar{\jmath}\bar{\imath}}$ where $\bar{\imath} =$ if $i\ mod\ 2 = 0$ then $i-1$ else $i+1$. Intuitively a cDBM does not change by switching positive with negative forms of the same variable. The switch operation $\bar{\imath}$ is typically implemented by a bit-wise `xor` operation. The other classic lattice operators are defined as follows:

$$\forall \mathbf{m}, \mathbf{n} \in \mathbf{cDBM}.\ (\mathbf{m} \sqcup^{\mathbf{cDBM}} \mathbf{n})_{ij} = \mathtt{max}(\mathbf{m}_{ij}, \mathbf{n}_{ij})$$
$$\forall \mathbf{m}, \mathbf{n} \in \mathbf{cDBM}.\ (\mathbf{m} \sqcap^{\mathbf{cDBM}} \mathbf{n})_{ij} = \mathtt{min}(\mathbf{m}_{ij}, \mathbf{n}_{ij})$$

The main result in the construction and representation of the octagon abstract domain is the existence of the best abstraction of octagons as an element in **cDBM**. This is achieved by computing *normal forms* for DBM representing octagons. A modified version of the Floyd-Warshall closure algorithm which performs *strong closure* is considered for this task. The intuition is that, while the Floyd-Warshall closure algorithm can be seen as a constraint propagation which completes a set of constraints until the following closure holds:

$$\begin{cases} V'_i - V'_k \leq a \\ V'_k - V'_j \leq b \end{cases} \Longrightarrow V'_i - V'_j \leq a + b$$

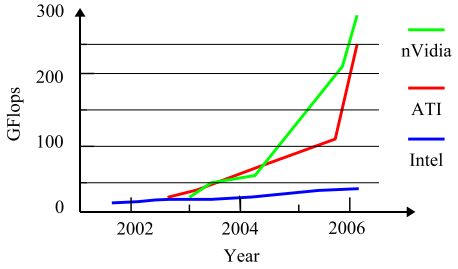

Fig. 1. A comparison of floating point performances between Intel CPU, ATI GPU and nVidia GPU in the last five years. As the graph shows GPU are increasing their performance every year faster than CPUs.

The modified Floyd-Warshall strong closure algorithm adds a second form of constraints until the following closure holds [18]:

$$\begin{cases} V'_{\bar{\imath}} - V'_i \leq a \\ V'_j - V'_{\bar{\jmath}} \leq b \end{cases} \Longrightarrow V'_j - V'_i \leq (a+b)/2$$

The constraints introduced by the (strong) closure algorithm are called *implicit* in order to distinguish them from the *explicit* constraints considered to build the octagon. The strong closure operation on DBM is denoted $(\cdot)^\bullet$. All the standard lattice-theoretic operations and C-like transfer functions have been defined on **cDBM** in order to derive an abstract semantics which has been proved correct by abstract interpretation [18].

3 An Overview on GPUs

In the last few years graphics hardware, known as GPU, dramatically increased its computationally power and flexibility for answering the need to increase the realism in videogames and other graphics applications. GPUs are quite a cheap product and they offer high performances, for example in the case of nVidia GeForce7950-GX2 (a double GPU equipped with 512Mb of RAM) the cost is around 310 euro (March 2007) offering a peak of 384 GFlops with a 51.2 Gb/sec bandwidth through the video memory, see figure Figure 2 for a complete sight on how a GPU is inserted in the traditional computer architecture. These performances are more than tripled compared with its predecessor the nVidia GeForce6800 Ultra, 58 GFlops with a 38.5 Gb/sec bandwidth. Indeed GPUs have an average yearly rate of growth around 2.0, which actually is higher than Moore's Law growth rate for CPU, 1.4 per year. See Figure 1 for the trend of GFlops in GPU in the last years.

3.1 The Programmable Graphics Pipeline

GPUs are optimized to render a stream of geometric primitives (point, lines and triangles), called *vertex buffer*, onto an array of pixels called *frame buffer*. The GPUs became in the last years fully programmable for transforming and lighting vertices and pixels. The main purpose of GPU is to processes vertices and pixels. This processing follows the classic graphics pipeline, allowing in certain stages

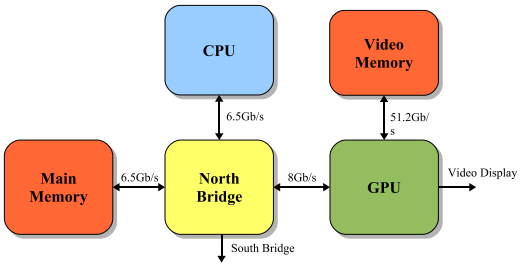

Fig. 2. The GPU in a traditional PC system: the GPU has a very large bandwidth with its memory (a peak of 51.2Gb/s in the nVida GeForce7900GTX). This value is nearly 8 times compared to the one for the CPU and its memory.

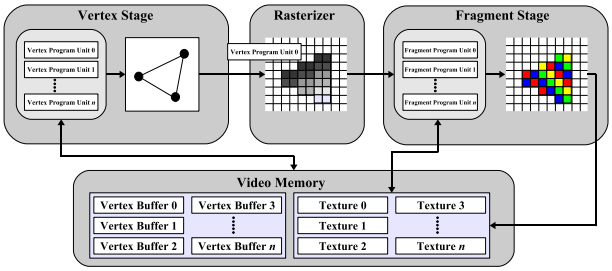

Fig. 3. The GPU Pipeline: vertices which define a primitive (triangle, point, line) are transformed using a function (implemented as a Vertex Program) in the Vertex Stage by Vertex Program Unit. Vertices are elaborated in parallel by many Vertex Program Units (Vertex Program Unit 0, ..., Vertex Program Unit n). Then primitives are discretized in fragments by the Rasterizer which passes these fragments to the Fragment Stage. At each fragment is applied a function (Fragment Program) in parallel using many Fragment Program Units (Fragment Program Unit 0, ..., Fragment Program Unit n). The result of the Fragment Stage is saved in a texture in the video memory.

programmability via micro programs. We can see a GPU pipeline in Figure 3. The first step in the pipeline is the vertices processing. Each vertex of a vertex buffer is processed by a Vertex Program Unit (VPU) which is a programmable unit that executes a vertex program (VP). A VP is a set of instructions that specifies how a vertex will be processed by the VPU. The output of a VPU is not only a modified position for a vertex but it can also add new properties to the vertex like a color, an address for fetching the memory during the next phase etc... A modern GPU presents more than one VPU (around 6) and it automatically distributes vertices to elaborate them fast between VPUs. Note that when a GPU is processing a single vertex buffer the VP is the same for all VPUs. After the VPU, a unit called rasterizer, generates fragments of the primitives, in other words it discretizes them in pixels and it interpolates values between vertices using linear interpolation. In the last step of the pipeline the fragments created by the rasterizer are processed by another programmable unit called Fragment Program Unit (FPU), which executes a fragment program (FP). As VP, the FP is a set of instructions which specifies how a fragment will be processed by the FPU. The output, which can be a single value or a vector of values, can be

stored in the frame buffer for visualization or in a texture for future processing. As for VPUs, FPUs are numerous in a modern GPU (around 32), and fragments are automatically distributed to FPUs. Note that as for the VP, the same FP is executed by all FPUs until all fragments generated by rasterizer are completed. In this current generation of GPUs the main instructions allowed in the VPs and FPs are: float point operations (addition, subtraction, division, multiplication, square root, logarithm, etc...), random access to the video memory, assignment command, static and dynamic branching (a costly operation), and loop (with limited loop size for avoiding infinite loops). VPs and FPs are usually written in a high-level programming language similar to C. These languages are called *shading languages* because they are designed for generating images. The most common shading languages are: C for Graphics (Cg) [16], the OpenGL Shading Language (GLSL) [13] and High Level Shading Language (HLSL) [1]. These languages provide an abstraction for a very close level to the hardware, indeed they manage directly vertices, textures, fragments, etc... which are very specific graphics primitives. Other languages present an higher abstraction avoiding direct manipulation of graphics primitives and supporting GP-GPU such as SH [17], BrookGPU [3], and etc... The main disadvantage of these higher abstractions is that they are implemented on top of Graphics API such as OpenGL and Direct3D, so the overhead is quite high.

3.2 Kernel Programming

Data parallelism is the key for high performance in GPUs. In this section, we shortly introduce the GPU programming model called Kernel Model. The most powerful components in the GPU architecture are FPUs, because they are more numerous than VPUs, usually in a ratio 6:1, allowing more parallel power. A GP-GPU program usually uses FPUs as main processing unit. The first algorithm is segmented into independent parallel parts, called kernels, each of these is implemented in a FPU. Inputs and outputs of each kernel are arrays of data, called texture, and they are stored in the video memory. A texture can be indexed in 1D (1D texture), in 2D (2D texture), and in 3D (3D texture). Note that 1D texture can have a size of only 4096 values, while 2D texture and 3D texture can allow a size up to respectively 4096^2 and 512^3 values. These are the following steps for kernel to run a kernel:

1. Vertices are passed to the GPU, in order to feed the vertex stage. A typical GP-GPU invocation is a quadrilateral, parallel to the screen of the display, which covers a region of pixels that match precisely with the desired size of the output texture. In our case this provides the extreme boundaries (a quadrilateral) of a temporary address space for allocating DBMs.
2. The rasterizer generates a fragment for every pixel in the quadrilateral. In our case the rasterizer fills the adress space with all the addresses, called fragments.
3. Each fragment is processed by the FPU. At this stage all the FPUs are processing the same fragment program. The fragment program can arbitrarily

```
float4 FP_Max(VertexOutput IN, Sampler2D texWork):
COLOR{float4
 ret= tex2D(texWork, IN.t0);
 ret=max(ret, tex2D(texWork,
          IN.t0+float2(iSize,0.0f)));
 ret=max(ret,tex2D(texWork,
          IN.t0+float2(0.0f,iSize)));
 return max(ret, tex2D(texWork,
          IN.t0+float2(iSize,iSize)));
}
```

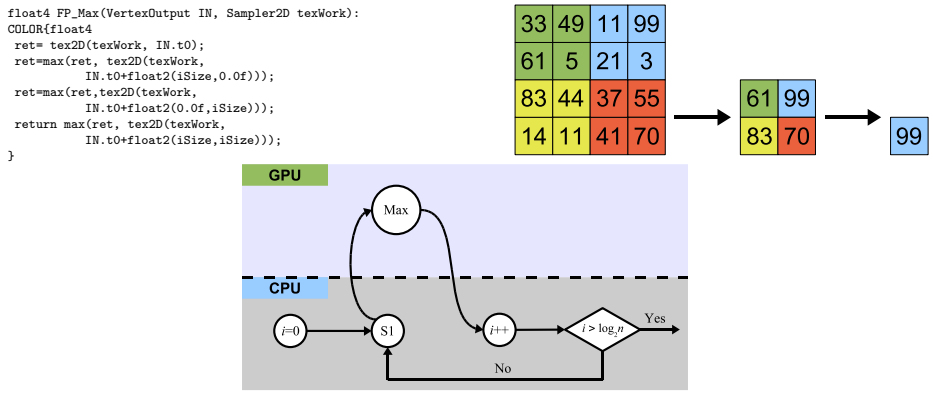

Fig. 4. The fragment program for calculating the maximum in a texture by reduction: IN is the input value of the FP calculated by interpolation from the rasterizer (IN.t0 is the vector that stores the coordinates of the current fragment), texWork is a 2D texture (declared as Sampler2D) in which are stored the values to reduce. The iSize is a constant value set in the FP, and represents the inverse of the size of the texture. tex2D is required to access to a texture in the video memory. Below the control flow of the calculation of maximum value using reduction paradigm. First the CPU sets the counter $i = 0$, then it enters in a loop. In the loop CPU transfers the control to GPU which calculates the maximum of every square of four pixels. After the GPU finishes it releases the control to CPU which increments i and it tests $i > log_2(n)$ where n is the width of the texture. If the guard is true the reduction is completed otherwise the CPU returns in S1.

read from textures in the video memory, but can only write to memory corresponding to the location of the fragment in the frame buffer determined by the rasterizer. The domain of the computation is specified for each input texture by specifying texture coordinates at each of the input vertices of the quadrilateral. In our case, the fragment program computes locally to a single pixel the operations in the octagon domain.

4. The output of the fragment program is a value, that is stored in a texture.

An algorithm needs to reiterate this process as many times as required by the computation. This is called *multi-pass*.

3.3 Reduction

A constraint of current GPU's generation is that a FPU cannot randomly write the result of its job in the video memory, but only on an address which is chosen by the GPU's rasterizer. This is a problem if we want to compute properties from a texture such as maximum, or minimum value. The solution is a process called Reduction: the kernel program gets as input the value of four neighbor pixels and computes the needed function using these four values. At the end it saves the output in a texture which is reduced by one half. This process is iterated

until the texture is only one single pixel; see Figure 4 for a visual example of the mechanism of reduction.

3.4 Addresses Issues in GPUs

In the current GPUs, integer arithmetic is not supported, only floating point is allowed. This feature will be present only in the upcoming GPU generation, R600 and GeForce Series 8 recently available. In particular when we want to access to a texture we need to use float addresses. The operator $\cdot^-$ is used in the octagon domain to access to $v_j^- = -v_j$ the negative variable. This operator is defined as $i^- \stackrel{\text{def}}{=} i$ `xor` 1. As we said before the XOR operation cannot be directly performed in the fragment. A solution to this problem is to encode in a single 1D texture, called XORTexture, all the XOR values for an address and save them in float values. Therefore, every time that we need to calculate a `xor` value of an address, we fetch the XORTexture with the address.

3.5 Float Precision Issues

GPUs can use two different types of floating point arithmetics: single precision, close to the 32-bit IEEE-754 standard, and half precision a 16-bit (10bits mantissa, 5bits exponent and 1bit sign). The main advantage of the half precision format is that a GPU performs nearly two times the speed of single precision, however they are not precise enough for numerical application, such as those employed in the octagon domain. We observed some numerical instabilities in the closure operation, therefore we decided to use the single precision format. This format has enough precision so there is no need to implement double precision in emulation using two single precision values (value and residual) [7]. GPUs present some issues with floating point arithmetic, because they do not implement the full IEEE-754 standard [12]. While GPUs can perform precise arithmetic operations (add, subtraction, multiplication, division, and tests) they cannot handle NaN value, and partially ±Inf. Also isnan and isinf functions are very dependent by the drivers of the vendors which usually strongly suggest to avoid them in fragment programs. This can be a problem in order to represent $\top^{\mathbf{cDBM}} = +\infty$ value in the octagon domain, however we simply solved this problem by assigning to $\top^{\mathbf{cDBM}}$ the value $3.4e38$, the maximum value representable in the GeForce series 6 and 7 architecture [22].

4 Mapping Octagon Abstract Domain on GPU

The Octagon Abstract Domain can be naturally mapped to GPUs because the data structure used to represent octagons, the DBM matrix, can be mapped one to one with a 2D texture of GPU. So there is no need to develop a particular data structure as in [15]. Another advantage is that the operators of the OAD are very simple matrix operators that can be performed by using simple kernel programming. In our implementation we did not use any API for GP-GPU. To

avoid overheads, we directly wrote the code by using OpenGL and Cg language for writing fragment programs.

4.1 Closure

In the octagon domain closure represents the normal form for the DBM, which calculates all the constraints (implicit and explicit) between the variables. This operation is defined in [18] by the following algorithm, which is a modified version of the Floyd-Warshall shortest-path algorithm:

$$\begin{cases} \mathbf{m}_0 \stackrel{\text{def}}{=} \mathbf{m} \\ \mathbf{m}_{k+1} \stackrel{\text{def}}{=} S(C_{2k}(\mathbf{m}_k)) \quad \forall k\colon 0 \le k \le n \\ (\mathbf{m})^{\bullet} \stackrel{\text{def}}{=} \mathbf{m}_n \end{cases} \tag{1}$$

where $\mathbf{m}$ is an empty octagon, C is defined as

$$[C_k(\mathbf{n})]_{ij} \stackrel{\text{def}}{=} \begin{cases} 0 & \text{for } i = j \\ \min\big(\mathbf{n}_{ij}, (\mathbf{n}_{ik} + \mathbf{n}_{kj}), \\ \qquad (\mathbf{n}_{i\bar{k}} + \mathbf{n}_{\bar{k}j}), \\ \qquad (\mathbf{n}_{ik} + \mathbf{n}_{k\bar{k}} + \mathbf{n}_{\bar{k}j}), \\ \qquad (\mathbf{n}_{i\bar{k}} + \mathbf{n}_{\bar{k}k} + \mathbf{n}_{kj})\big) & \text{elsewhere} \end{cases} \tag{2}$$

and S as

$$[S(\mathbf{n})]_{ij} \stackrel{\text{def}}{=} \min(\mathbf{n}_{ij}, (\mathbf{n}_{i\bar{i}}\mathbf{n}_{\bar{j}j})/2) \tag{3}$$

This operation can be implemented on a GPU in the following way. First the

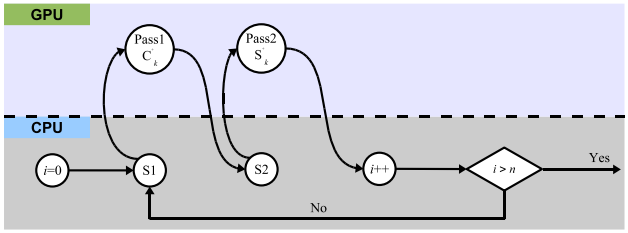

Fig. 5. The Closure control flow: at each cycle the CPU set parameters for the fragment program PS_Closure_C in S1 (k, and the working texture T_0), then the GPU executes PS_Closure_C saving the result in a texture, T_1. The CPU gets back the control and at stage S2 sets the parameter for fragment program PS_Closure_S (the working texture T_1), which is then executed by the GPU. Again the CPU gets back the control and increases the variable i. If $i > n$ (where n is the number of variables in the program) the closure is reached, otherwise the CPU reiterates S1.

texture, T_0, representing the octagon that we want to close is applied C using FP_CLOSURE_C in Figure 6, and it is saved in the video memory in another texture, T_1. Secondly at T_1 is applied S using FP_CLOSURE_S in Figure 6 and the result is saved in T_0, see Figure 5 for a visualization of the control flow. This process is iterated n times, where n is the number of variables in the program.

```
float4 FP_Closure_C(vertexOutput IN, Sampler2D texWork_T1): COLOR{
    float2 c0= IN.t0.xy;
    if(c0.x==c0.y)
        return 0.0f;
    else{
        float4 val0,val1,tmpVal,tmpVal2,tmpVal3,tmpVal4;
        tmpVal=tex2D(texWork1,float2(k,c0.y));
        tmpVal2=tex2D(texWork1,float2(c0.x,kXOR));
        tmpVal3=tex2D(texWork1,float2(kXOR,c0.y));
        tmpVal4=tex2D(texWork1,float2(c0.x,k));
        val0=    tex2D(texWork1,c0);
        val1=    tmpVal4+tmpVal;
        val0=(val0>val1)?val1:val0;
        val1=    tmpVal2+tmpVal3;
        val0=(val0>val1)?val1:val0;
        val1=    tmpVal4+tex2D(texWork1,float2(k,kXOR))+tmpVal3;
        val0=(val0>val1)?val1:val0;
        val1=    tmpVal2+tex2D(texWork1,float2(kXOR,k))+tmpVal;
        return (val0.x>val1.x)?val1.x:val0.x;
    }};
```

```
float4 FP_Closure_S(vertexOutput IN, Sampler2D texWork_T1):
COLOR{
    float2 c0=IN.t0.xy;
    float4 val0,val1;
    val0=(tex2D(texWork_T1,float2(c0.x,XORAddress(c0.x)))+
     tex2D(texWork_T1,float2(XORAddress(c0.y),c0.y)))/2.0f;
    val1=tex2D(texWork_T1,c0);
    float4 ret=(val1<val0)?val1:val0;
    return ret>1e30?3.4e38f:ret;
};
```

Fig. 6. FP_Closure_C is the FP that implements the C function: values k and kXOR are constant values set by CPU, kXOR is the xorred value of k. k represents the value k in Equation 2. FP_CLOSURE_S implements S (Equation 3).

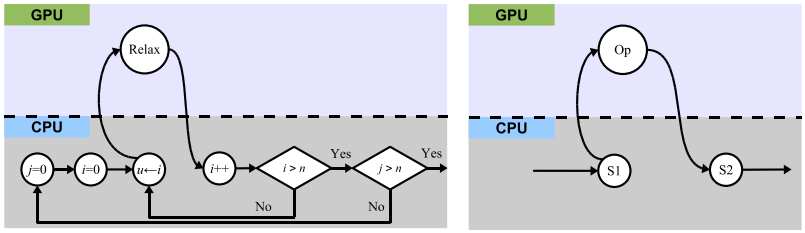

Fig. 7. GPU and CPU control flow: a) the control flow for the relaxation part of the Bellman-Ford Algorithm. b) the control flow of a simple operation, the CPU leaves the control to the GPU to execute the fragment program for the simple operation.

4.2 Emptiness Test

The emptiness test checks if an octagon $(\mathbf{m})^\bullet = \emptyset$. This happens if only if the graph of $(\mathbf{m})^\bullet$ has a cycle with a strictly negative weight [18]. A well known algorithm for detecting negative weight cycle is the Bellman-Ford algorithm.
The mapping is realized as follow. Firstly we initialize texDist the distance array, a 1D Texture, using a constant shader which returns infinity (3.4e38). The size of this texture is $2n$, where n is the number of variables. Secondly we compute the first part of the algorithm, we iterate $(2n)^2$ the FP_Relax, Figure 8, that relaxes edges, see Figure 7.a for control flow. Finally we call FP_Relax_Red, Figure 8, to find out if there is negative cycle. This function marks with value 1.0 a negative cycles, otherwise it return 0.0. The result of the test is collected applying a reduction using a Maximum kernel Figure 4.

4.3 Simple Operators

There are some operators used in the octagon domain that require only one pass to obtain the result. This means that the fragment program for that operator is

```
float4 FP_Relax(vertexOutput IN, float u,
                Sampler1D texDist, Sampler2D texWork1): COLOR{
    float4 distU=tex1D(texDist,u);
    float4 distV=tex1D(texDist,IN.t0.y);
    float4 weight=tex2D(texWork1,float2(u,IN.t0.y));
    if(weight>=3.4e38)
        return distV;
    else{
      float4 sum=distU+weight;
      return distV>sum?sum:distV;
    }};
```

```
float4 FP_Relax_Red(vertexOutput IN,
                    Sampler1D texDist, Sampler2D texWork1): COLOR{
    float4 distU=tex1D(texDist,IN.t0.x);
    float4 distV=tex1D(texDist,IN.t0.y);
    float4 weight=tex2D(texWork1,IN.t0.xy);
    if(weight>=3.4e38)
       return 0.0;
    else
       return (distU>=(distV+weight))?1.0:0.0;
};
```

Fig. 8. FP_Relax is a FP that implements the relaxation for a row in the adjacency matrix. FP_Relax_Red is a FP that checks if there is a negative cycle (returns 1.0), the complete check is realized applying a max reduction operation.

called only once, in Figure 7.b the control flow between CPU and GPU is shown for simple operators. These operators are: union, intersection, test guard, and assignment.

Union and Intersection. Union and Intersection between octagons are both used to implement complex guards and to merge the control flow in *if else* and *loop* commands. These operators are implemented using the upper $\sqcup^{\mathbf{cDBM}}$ and lower $\sqcap^{\mathbf{cDBM}}$ bound operators [18]:

$$[\mathbf{m} \sqcap^{\mathbf{cDBM}} \mathbf{n}]_{ij} \stackrel{\text{def}}{=} \min(\mathbf{m}_{ij}, \mathbf{n}_{ij}) \tag{4}$$

$$[(\mathbf{m})^{\bullet} \sqcup^{\mathbf{cDBM}} (\mathbf{n})^{\bullet}]_{ij} \stackrel{\text{def}}{=} \max((\mathbf{m})^{\bullet}_{ij}, (\mathbf{n})^{\bullet}_{ij}) \tag{5}$$

The implementation on GPU is quite effortless, it is only required to write in the fragment program Equation 4 and Equation 5, as it is shown in Figure 7. Note that when we calculate the union operator we need to apply the closure to $\mathbf{m}$ and $\mathbf{n}$.

Test Guard Operator. The test guard operator model how to analyze guards in programs. The main guard tests, that can be modeled in the octagon domain, are: $v_k + v_l \leq c$, $v_k - v_l \leq c$, $-v_k - v_l \leq c$, $v_k + v_l = c$, $v_k \leq c$, and $v_k \geq c$. All these various tests can be similarly modeled by using the first test, as proved in [18]. So we will illustrate the implementation for $v_k + v_l \leq, c$ the others are similar. The octagon operator for this is defined as:

$$[\mathbf{m}_{(v_k+v_l\leq c)}]_{ij} \stackrel{\text{def}}{=} \begin{cases} \min(\mathbf{m}_{ij}, c) & \text{if } (j,i) \in \{(2k, 2l+1); (2l, 2k+1)\} \\ \mathbf{m}_{ij} & \text{elsewhere} \end{cases} \tag{6}$$

In this case, as for Union and Intersection operators, we need only to write a simple fragment program that implements Equation 6. However in order to save very costly *if-else* commands, for checking if $(j,i) \in \{(2k, 2l+1); (2l, 2k+1)\}$, we can solve this calculating the dot product between the difference vector of $\{(2k, 2l+1), (2l, 2k+1)\}$ and (j,i). This operation could be heavy, but dot product is an hardware built-in function and it performs faster than executing *if-else* commands on a GPU [22], see Figure 7 for the fragment program on GPU for this operator.

Assignment Operators. The assignment operators model how to analyze assignments in programs. The main assignments, that can be modeled in the octagon domain, are: $v_k \leftarrow v_k + c$, $v_k \leftarrow v_l + c$ and $v_k \leftarrow e$ where e is a generic expression. As for the test guards operators we will show the implementation on the GPU for the first assignment, the others are similar. Firstly we define the assignment operator for $v_k \leftarrow v_k + c$:

$$[\mathbf{m}_{(v_k \leftarrow v_k + c)}]_{ij} \stackrel{\text{def}}{=} \mathbf{m}_{ij} + (\alpha_{ij} + \beta_{ij})c \tag{7}$$

with

$$\alpha_{ij} \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } j = 2k \\ -1 & \text{if } j = (2k+1) \\ 0 & \text{elsewhere} \end{cases} \qquad \beta_{ij} \stackrel{\text{def}}{=} \begin{cases} -1 & \text{if } i = 2k \\ 1 & \text{if } i = (2k+1) \\ 0 & \text{elsewhere} \end{cases} \tag{8}$$

Again as in the case of the Union or Intersection operators we need only to write Equation 7 in the fragment program, as it is shown in Figure 9.

```
float4 FP_ASSIGN1(vertexOutput IN, Sampler2D texWork1, float c,
                  float k2, float k2add1): COLOR{
float4 val=tex2D(texWork_T1,IN.t0);
float alpha,beta;
if(IN.t0.y==k2)
alpha= 1.0f;
else
alpha= (IN.t0.y==k2add1)?-1.0f:0.0f;
if(IN.t0.x==k2)
beta=-1.0f;
else
beta= (IN.t0.x==k2add1)?1.0f:0.0f;
return val+(beta+alfa)*c;};
```

```
float4 FP_TEST_GUARD1(vertexOutput IN, Sampler2D texWork1,
     float c, float2 coord1, float coord2): COLOR{
float4 val=tex2D(texWork_T1,IN.t0);
float2 diff;
float ret;
diff=IN.t0.yx-coord1;
if(dot(diff,diff)==0.0f)
return min(val,c):val;
diff=IN.t0.yx-coord2;
return (dot(diff,diff)==0.0f)?min(val,c):val;};
```

Fig. 9. The fragment program for the assignment $v_k \leftarrow v_k + c$ and test guard $v_k + v_l \leq c$.

4.4 Widening

Widening is an operator that is used to speed-up the convergence in abstract interpretation [4] returning an upper approximation of the least fixpoint $\bigvee_{i \in \mathbb{N}} F^i(\mathbf{m})$ greater than $\mathbf{m}$ of an operator (predicate transformer) F:

$$[\mathbf{m} \nabla \mathbf{n}]_{ij} \stackrel{\text{def}}{=} \begin{cases} \mathbf{m}_{ij} & \text{if } \mathbf{n}_{ij} \leq \mathbf{m}_{ij} \\ +\infty & \text{elsewhere} \end{cases} \tag{9}$$

As it can be seen from Equation 9, the widening operator can be easily realized as a simple operator, indeed the fragment is very simple, see Figure 11. When we analyze a loop such as:

$$[l_i \textbf{ while } g \textbf{ do } l_j ... l_k \textbf{ done } l_{k+1}]$$

where l_i is a pointer to a program location we need to solve $\mathbf{m}_j = (\mathbf{m}_i \sqcup^{\mathbf{cDBM}} \mathbf{m}_k)_g$, this is done iteratively. Starting from $\mathbf{m}_i$, the octagon for location l_i, $\mathbf{m}_k$ can be deduced from any $\mathbf{m}_j$ using propagation. We compute the sequence $\mathbf{m}_j$:

$$\begin{cases} \mathbf{m}_{j,0} = (\mathbf{m}_i)_{(g)} \\ \mathbf{m}_{j,n+1} = \mathbf{m}_{j,n} \nabla ((\mathbf{m}_i)^{\bullet}_{(g)}) \end{cases} \tag{10}$$

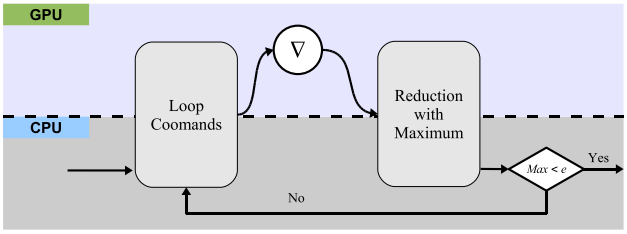

Fig. 10. The widening control flow: when we need to analyze a loop we proceed iteratively to the calculation of the fix point for that loop, this process needs to analyze commands in the loop, *Loop Commands* box in the flow, then we calculate the widening between $\mathbf{m}_{j,n}$ and $((\mathbf{m}_i)^{\bullet}_{(g)})$. After that we check if $\mathbf{m}_{j,n}$ and $\mathbf{m}_{j,n+1}$ represent the same octagon. This achieved calculating the maximum of the difference of them using the reduction paradigm performed in the *Reduction with Maximum* box. If the maximum is lower than e a small positive value they are the same octagon.

and finally $\mathbf{m}_{k+1}$ is set equal to $((\mathbf{m}_i)^{\bullet}_{\neg g}) \sqcup^{\mathbf{cDBM}} ((\mathbf{m}_k)^{\bullet}_{\neg g})$. The calculation of the whole widening process on GPU for analyzing loops is performed in the following way: we enter in the loop and we analyze each command in the loop. After that we calculate the widening on GPU using Figure 11 between $\mathbf{m}_{j,n}$ and $((\mathbf{m}_i)^{\bullet}_{(g)})$. Finally we check if we have reached a fix point, this is realized by comparing $\mathbf{m}_{j,n+1}$, widening result, with $\mathbf{m}_{j,n}$. If it is the same octagon we reached the fix point, otherwise we need to reiterate the process. This comparison on GPU is achieved calculating the difference, D_0 between the result of the widening and the previous result. At this point we calculate the maximum value of D_0 using reduction paradigm, if the maximum is lower then a certain threshold e (a small value greater than zero) the two octagons are the same otherwise they are different, all these operations can be seen summarized in Figure 10.

```
float4 FP_Widening(vertexOutput IN, Sampler2D texWork1): COLOR{
        float4 r1=tex2D(texWork1, IN.t0);
        float4 r2=tex2D(texWork2, IN.t0);
        return r2<r1?r1:3.4e38;
};
```

```
float4 FP_Top(vertexOutput IN): COLOR{
        return (IN.t0.x==IN.t0.y)?3.4e38f:0.0f;};

float4 FP_Bottom(vertexOutput IN): COLOR{
        return 0.0f;};
```

Fig. 11. The widening operator and the basic octagons $\top^{\mathbf{cDBM}}$ and $\bot^{\mathbf{cDBM}}$.

4.5 Packing Data in RGBA Colors

Current GPUs can handle 4096×4096 2D texture size, so the maximum number of variables is 4096. However we can allow 8192 variables using pixel packing. A pixel is generally composed by four components: red, blue, green and alpha. So we can easily use these channels to allow bigger matrices, this means we treat red, green, blue, and alfa as four neighbor values in the octagon, see Figure 4.5. One advantage of RGBA packing is that we do not have to modify our

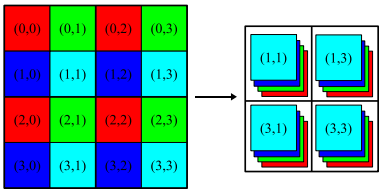

Fig. 12. The RGBA Packing: four neighbor values are packed in a single pixel using red, green, blue and alfa channels.

fragment programs, since GPU performs vector float point arithmetic operators, assignments, and the ternary operator (test?cond1:cond2). Also we do not use the values of a octagon in the dynamic branching (*if-else*), therefore we do not have to extend the branching for each components. Another common technique is to flatten 3D textures, this means to access to a (i, j, k) memory location using a (i, j) address. However representing octagon bigger than 8192×8192 using 3D texture implies to use more than 64MB of video memory for each, so we can keep on the GPU system only few octagons.

4.6 Static Analyzer

We designed a (naive) static analyzer as presented in [6, 18], in which the control flow is performed by the CPU but simple operators (union, intersection, assignments, guard checks), closure, widening, and fix point check are performed on GPU. When the interpreter starts to analyze a program, first it sets $\mathbf{m} = \top^{\mathbf{cDBM}}$ the octagon associated with the first program point, by using the simple fragment program in Figure 11. Then it interprets programs naively by applying octagon operators for the various commands and combinations of them:

- $[\ l_i\ v_i \leftarrow e\ l_j\ ]$: we used the assignment operators;
- $[\ l_i$ **if** g **then** $l_j ...$ **else** $l_k ...$ **end if** $l_p\ ]$: for the branch l_j we apply the guard operator g to the octagon $\mathbf{m}_i$) (representing point l_i), while for the branch l_j we apply the guard operator $\neg g$ to the octagon $\mathbf{m}_i$). When the flow control merges at point l_p we use the union operator:

$$\mathbf{m}_p = ((\mathbf{m}_j)^\bullet) \sqcup^{\mathbf{cDBM}} ((\mathbf{m}_k)^\bullet) \tag{11}$$

- $[\ l_i$ **while** g **do** $l_j ... l_k$ **done** $l_{k+1}\ ]$: we used the widening operator as presented in Section 4.4 for approximating the fixpoint.

5 Results

We implemented our abstract interpreter in C++, by using OpenGL [20], a successful API for computer graphics that allows GPU programming, and Cg language for GPU programming. For results we used a machine equipped with an Intel Pentium 4 D 3.2 Ghz processor, 2GB of main memory, and a GeForce 7950-GTX with 512MB of video memory. We compared the results of our interpreter with a single threaded CPU interpreter. In our experiments, with randomly generated octagons, we timed single operators such as reduction, closure,

intersection, union, assignment, test guard, widening, and emptiness test. The results for these operators by using a single GPU, compared with a CPU, are presented in Figure 13. For each operators we reached the following speed-ups,

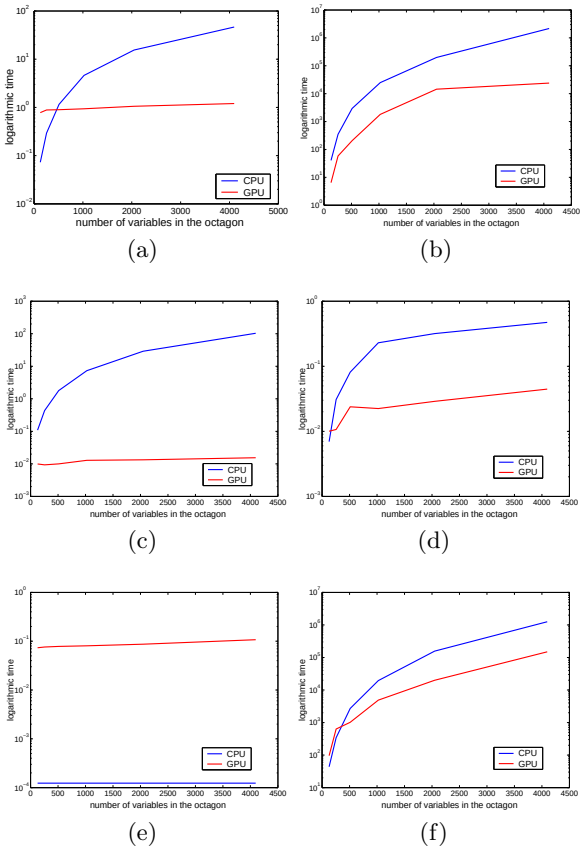

Fig. 13. Timing comparison between CPU and GPU for the octagon domain operators. We displayed timing in logarithmic scale of the time expressed in milliseconds: a) reduction operator (for checking fix point). b) closure operator. c) union (similar for intersection and widening). d) assignment. e) test guard. f) emptiness test.

that can be derived from Figure 13:

1. **Reduction Operator**: while the computational complexity for the CPU implementation is $\mathcal{O}(n^2)$, the one for GPU is $\mathcal{O}(\frac{n^2}{p}\log n^2)$, where n is the number of variables in an octagon and p is the number of FPUs. However its computational constant is lower than the one for CPU, so the speed-up is reasonable, achieving 9.98 times in average.

2. **Closure Operator**: the computational complexity is the same for both implementations, GPU and CPU, $\mathcal{O}(n^3)$. We achieved a 24.13 times speed-up in average.
3. **Emptiness Test**: the computational complexity is the same for both implementations, GPU and CPU, $\mathcal{O}(n^3)$. We achieved a 4.0 times speed-up in average. Note that the speed-up is lower than the one for the Closure Operator because we need to perform a test (FP_Relax_Red) and a reduction operation.
4. **Union Operator**: the computational complexity is the same for both implementations, GPU and CPU, $\mathcal{O}(n^2)$. We achieved a 160.5 times speed-up in average.
5. **Assignment Operator**: while the computational complexity of this operator for the CPU implementation is $\mathcal{O}(n)$ (we need to modify only two columns), the one for GPU is $\mathcal{O}(n^2)$. This is caused by SIMD nature of GPUs that needs to work on all values of a texture and not only a portion of it. For this operator we reached a lower speed-up than other operators, 6.47 times in average.
6. **Test Guard Operator**: this represents the worst operator in our implementation test. Since for the CPU implementation we need to modify only a constant number of values in the octagon, its computational complexity is $\mathcal{O}(1)$. However on GPUs we cannot modify only few values in a texture, but all values, so the complexity is $\mathcal{O}(n)$. In this case the CPU performs better than GPU with a 175.43 times speed-up in average. However it is faster to perform this operator on GPU than a computation on CPU followed by a transfer to GPU. This is because we need to transfer a big amount of data through the bus, which is typically a bottleneck in the architecture (see Figure 2).
7. **Widening Operator**: as in the case of Union Operator, the computational complexity for both implementations is $\mathcal{O}(n^2)$. We achieved a 114.7705 times speed-up in average.

Number of variables	Reduction	Closure	Union	Assignment	Test guard	Widening	Emptiness Test
CPU							
128	0.072928	40.42	0.095516	0.0069554	0.0001233	0.10912	43.73
256	0.29318	348.79	0.36542	0.030791	0.0002158	0.43517	342.10
512	1.1579	2949.90	1.5013	0.081164	0.0002226	1.7894	2716.42
1024	4.618	2.4863e4	6.8319	0.22949	0.001927	7.3287	1.9501e4
2048	15.491	1.9902e5	24.402	0.31771	0.0001477	28.997	1.5683e5
4096	46.444	2.172e6	73.212	0.47234	0.0001477	103.17	1.2546e6
GPU							
128	0.77948	6.4472	0.08497	0.010033	0.073362	0.099478	95.947
256	0.88522	58.346	0.097702	0.010693	0.076067	0.093238	632.03
512	0.89636	206.1	0.104	0.02383	0.0782	0.10923	1019.10
1024	0.93726	1804	0.1093	0.022384	0.080448	0.1288	4880.9
2048	1.0562	1.4470e4	0.11889	0.0289	0.086864	0.13345	1.9912e4
4096	1.2043	2.3951e4	0.18231	0.044535	0.10735	0.15447	1.4934e5

From the Table above the results of our implementation on CPU and GPU, the timing is expressed in second for 20 runs for each operator. As can we see, we obtain a sensible speed-up for simple operators (intersection, union, assignment, test guard, widening, emptiness test) and a speed-up around 24 times for closure operator. The bottleneck in our system is given by the test guard operator (Figure 13.e), indeed an optimized CPU implementation of this operator takes only $\mathcal{O}(1)$.

6 Conclusion and Future Work

We presented a new implementation of Abstract Octagon Domain on GPU, improving efficiency in time. Another advantage of our implementation is that it is fully compatible with old GPUs (3-4 years models) and not only new models. Computational complexity of the algorithm is kept the same, with exception of the operation for checking if the fix point has been reached during the analysis of loops. While the complexity of this operation on CPU is $\mathcal{O}(n^2)$ where n is the number of variables, it is $\mathcal{O}(\frac{n^2}{p} \log n^2)$, where p is the number of FPUs, due to the overhead in the reduction phase. The main limits of our current implementation are: the test guard operator which is linear in the number of variables and the size of a DBM, which is 8192×8192, meaning that we can model octagons with 4096 variables using the RGBA packing technique. In future work we would like to extend the size of octagons using hierarchical techniques as presented in [15]. In these techniques, a larger texture is sliced in subtextures which are addressed using a *page texture*. A page texture presents as values pointers to access to the desired subtexture. We are also interested in upgrading our implementation with upcoming Graphics Hardware. One of the main advantages of the new generation is the ability to randomly write the results of a fragment program in the video memory. Therefore there will be no more need to perform reduction to check when to stop in the widening operator or for the emptiness test, improving the complexity and performance for these operators, which represents our main bottleneck. Another advantage would be the suppression of XORAddress function, since new GPUs present integer arithmetic, saving memory and GPU performance. The new generation presents a better floating point implementation (very close to IEEE754 standard, with still some issues for handling specials) that could improve our implementation. This improved precision does not solve the unsound problems, that can be solved using interval linear forms [19]. In future work we would like to map efficiently interval linear forms on GPU.

References

1. D. Blythe. The direct3d 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.
2. J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.
3. I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
5. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP'05*, volume 3444 of *LNCS*, pages 21–30. Springer, April 2005.

6. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96, New York, NY, USA, 1978. ACM Press.
7. G. Da Graçca and D. Defour. Implementation of float-float operators on graphics hardware. *ArXiv Computer Science e-prints*, March 2006.
8. 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*, page 47, Washington, DC, USA, 2004. IEEE Computer Society.
9. N. Galoppo, N. Govindaraju, M. Henson, and D. Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.
10. N. Govindaraju, J. Gray, R. Kumar, and D. Manocha. Gputerasort: high performance graphics co-processor sorting for large database management. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336, New York, NY, USA, 2006. ACM Press.
11. D. Reiter Horn, M. Houston, and P. Hanrahan. Clawhmmer: A streaming hmmer-search implementation. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 11, Washington, DC, USA, 2005. IEEE Computer Society.
12. IEEE. Ieee 754: Standard for binary floating-point arithmetic.
13. J. Kessenich, D. Baldwin, and R. Rost. The opengl shading language v.1.20 revision 8. September 2006.
14. J. Kruger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 908–916, New York, NY, USA, 2003. ACM Press.
15. A. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. Owens. Glift: Generic, efficient, random-access gpu data structures. *ACM Trans. Graph.*, 25(1):60–99, 2006.
16. W. Mark, R. Glanville, K. Akeley, and M. Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, pages 896–907, New York, NY, USA, 2003. ACM Press.
17. M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 787–795, New York, NY, USA, 2004. ACM Press.
18. A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.
19. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP'04*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.
20. J. Neider and T. Davis. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.
21. J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krger, A. Lefohn, and T. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.
22. M. Pharr and R. Fernando. *GPU Gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley Professional, March 2005.
23. S. Venkatasubramanian. The graphics card as a stream computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003.