High-Performance VNS for the Max-Cut Problem using Commodity Graphics Hardware

Antonio S. Montemayor, Abraham Duarte, Juan José Pantrigo, Raúl Cabido

Universidad Rey Juan Carlos
{antonio.sanz, abraham.duarte, juanjose.pantrigo, raul.cabido}@urjc.es

Abstract

The Max-Cut problem consists of finding a partition of the graph nodes into two subsets, such that the sum of the edge weights having endpoints in different subsets is maximized. This NP-hard problem for non planar graphs has different applications in areas such as VLSI and ASIC design. In the last decade, consumer graphics cards (GPUs) have increased their power due to the computer games industry. These cards are now programmable and capable of processing huge amounts of data in a Streaming Pipelined Architecture. This paper proposes a high-performance GPU implementation of Variable Neighborhood Search (VNS) for the Max-Cut problem. This algorithm is tested and compared with the non-parallel implementation of VNS in CPU.

Keywords: VNS, Max-Cut, Graphics Hardware, Parallelization.

1 Introduction

An important graph bipartition problem is the Max-Cut problem defined for a undirected weighted graph $S = (V, E, W)$, where V is the set of vertices or nodes ($|V| = n$), E is the set of undirected arcs or edges ($|E| = m$), and W is the set of edge weights. The Max-Cut optimization problem consists of finding a partition of the set V into two disjoint subsets (C, C') such that the sum of the weights of edges with endpoints in different subsets is maximized. Every partition of vertices V into C and C' is usually called a cut or cutset and the sum of the involved edges weights is called weight of the cut or cut value. The considered Max-Cut optimization problem is given by the maximization of the cut value:

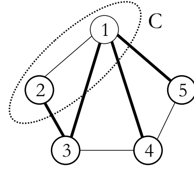

Figure 1: Cut example (C) for an undirected graph.

$$s(C, C') = \sum_{v \in C, u \in C'} w_{vu}$$

where w_{uv}, is the weight of edge $(u, v) \in V$. Reference [7] proves that the decision version of Max-Cut problem is NP-Complete. Therefore, it is convenient to devise algorithms for finding an approximate solution to this problem in a reasonable time. Figure 1 shows a cut example for an undirected graph. The cut edges are represented with thick lines. Assuming that all edges have the same weight, that is equal to one, the cut value shown in this example is 4. Some practical applications of the Max-Cut problem can be found in diverse fields like VLSI design, statistical physics and other related to combinatorial optimization. Several continuous, linear programming and semidefinite relaxations for the Max-Cut have been proposed to achieve high quality solutions in a reasonable running time. Among these alternatives, the most suitable is the semidefinite relaxation (SDP) because it is solvable in polynomial time. Moreover, this SDP value establishes an upper bound of optimal cut values [1]. It can be used to test the performance of approximate algorithms for the referred problem. Goemans et al. [3] proposed a randomized algorithm that guarantees a 0.878-approximation to the optimum and, in addition, an upper bound on the optimum. A very interesting rank-2 relaxation algorithm is proposed in [2] that gives, in mean, better solutions than other theoretical relaxations[1].

Apart from that, multimedia and computer games industry have encouraged graphics hardware to improve their processing power to unprecedent limits. Their processing power should not be underestimated and many authors have demonstrated that these consumer Graphics Processing Units (GPU) have a great raw performance, even superior to the most common and powerful CPUs [8, 6]. The architectural design of graphics cards makes them very scalable, and its performance is doubled every 6-9 months, much faster than CPUs. But even more interestingly, these GPUs can be programmed to customize their rendering pipeline and thus generating personalized special effects.

Besides, developers can take advantage of these programmable capabilities even with applications far beyond rendering purposes. In this way, the GPU becomes a co-processor for the central processing unit (CPU) with the idea that they can be encountered in most off-the-shelf desktop computers. Examples that demonstrate this fact are found in applications that exploits the power of the GPU for linear algebra calculations [8], physically-based simulations [6] or neural network implementations [10], among many others [5].

In this paper we propose a high-performance implementation of Variable Neighborhood Search (VNS) [4] metaheuristic for finding an approximate solution to the Max-Cut problem. For the implementation we use a common graphics card as a computational intensive processor. In order to evaluate the performance of our approach, we compare results produced by our algorithm with a non-parallel implementation on the same standard benchmarks.

2 VNS for the Max-Cut Problem

This section resumes the main features of Variable Neighborhood Search (VNS) metaheuristic. This metaheuristic, which was originally proposed by Hansen and Mladenović [4], is based on the exploration of a dynamic neighborhood model. Each step has three major phases: neighbor generation, local search and jump.

Unlike other metaheuristics based on local search methods, VNS allows changes of the neighborhood structure during the search. VNS explores increasingly neighborhoods of the current best found solution x. The basic idea of VNS is to change the neighbourhood structure when the local search is trapped on a local minimum.

Let $N_k, k = 1, ..., k_{max}$ be a set of predefined neighborhood structures and let $N_k(x)$ be the set of solutions in the kth-order neighborhood of a solution x. In the first phase, a neighbor $x' \in N_k(x)$ of the current solution is applied. Next, a solution x'' is obtained by applying local search to x'. Finally, the current solution jumps from x to x'' if it improves the former one. Otherwise, the order $k$of the neighborhood is increased by one and the above steps are repeated until some stopping condition is met. The pseudo-code of a typical VNS procedure is illustrated in Figure 2 a.

In the case of the Max-Cut problem, the kth-order neighborhood is defined by all solutions that can be derived from the current one by selecting k vertices and transferring each vertex from one

```
procedure VNS(x)
  var
    x: Initial solution
    x',x'': Intermediate solutions
    k: Neighbourhood order
  begin
  /*First Neighbourhood Structure*/
  k = 1;
  while k < kmax do
    /*Select an random solution in k-
    neighbourhood structure*/
    x'  = Random(x,Nk(x))
    /*Use the local search procedure shown
    in Figure 3*/
    x'' = LocalSearch(x');
    /*Replace the actual solution by the new
    one when an improvement is obtained */
    if w(x'') > w(x) then
      x = x'';
      k = 1;
    else
      k = k + 1;
    end if
  end while
  end
 end VNS
```

(a)

```
procedure Local_Search(g)
  var
    g=(C,C´): Cutset structure
  begin
  for v = 1 to Nodes_in_considered_graph do
    if v∈C and σ(v)>σ'(v) then
      /* v: C→C´ */
       C = C \ {v};
       C´= C' ∪ {v};
    end if
    if v∈C´and σ(v)<σ'(v) then
      /* v: C'→C */
      C'= C'\{v};
      C = C ∪ {v};
    end if
  end for
  end
 end Local_Search
```

(b)

Figure 2: (a) VNS high level pseudo-code. (b) Local search high level pseudo-code.

subset of the vertex bipartition to the other subset. The local search phase is based on the following neighborhood structure. Let $(C_a,\, C'_a)$ be the current cutset solution. For each vertex $v \in V$ we associate a new neighbor cutset $(C_b,\, C'_b)$:

$$(C_b, C'_b) = N_1(C_a, C'_a) = \begin{cases} C_b = C_a - \{v\}; C'_b = C'_a + \{v\} & if \quad v \in C_a \\ C_b = C_a + \{v\}; C'_b = C'_a - \{v\} & if \quad v \in C'_a \end{cases}$$

We define for each node $v \in V$ the functions $\sigma(v)$ and $\sigma'(v)$ as:

$$\sigma(v) = \sum_{u \in C} w_{vu} \quad \sigma'(v) = \sum_{u \in C'} w_{vu}$$

These two functions are characterized by the change in the objective function value associated with moving vertex v from one subset of the cut to the other. This way, a vertex makes a movement in order to improve the cut value in the two following situations:

$$\begin{aligned} &if \quad v \in C \quad \wedge \quad \sigma(v) > \sigma'(v) \quad then \quad C \longrightarrow C' \\ &if \quad v \in C' \quad \wedge \quad \sigma'(v) > \sigma(v) \quad then \quad C' \longrightarrow C \end{aligned}$$

where $C \longrightarrow C'$ (equivalently $C' \longrightarrow C$) represents the movement of vertex v from subset C to C' $(C \cup C' = V)$.

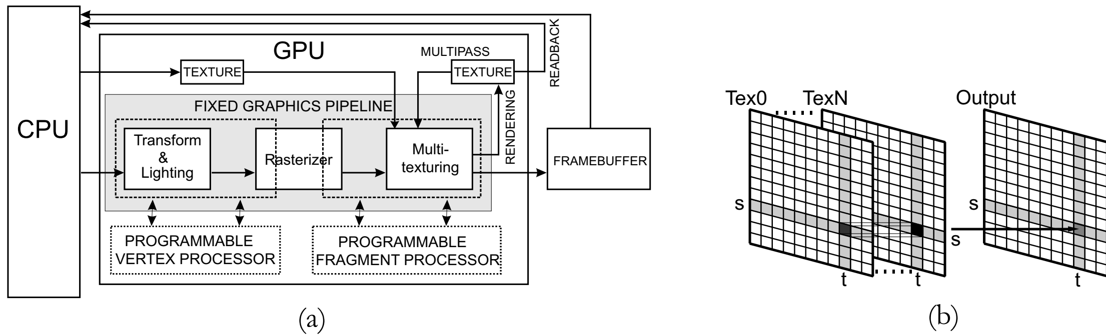

Figure 3: (a) Basic CPU/GPU programming model. When enabled, programmable vertex and fragment execution paths replace their corresponding stages of the fixed graphics pipeline (represented in dot-lines). (b) Simple fragment program computation. A common fragment program will be executed over every position of the input textures ($Tex0$-$TexN$), for example returning a resulting value at (s,t) of the output texture for values at (s,t) of the input ones.

All possible moves are examined. The current solution is replaced by its best improving neighbor solution. The search stops after all possible moves have been evaluated and no improving neighbor is found. The used local search strategy is summarized by the pseudo-code of Figure 2 b.

This local search procedure tests all possible movements for each node between C and C' and vice versa. Therefore, the current solution is replaced by the best solution found in the neighborhood structure defined above. The procedure ends when none possible neighbor movement improves the current solution.

3 Graphics Hardware

Commodity graphics hardware has evolved drastically since the mid 90's. With the aid of the rapid expansion of computer games and multimedia technologies these consumer GPUs have also become very powerful and inexpensive hardware. Traditionally, these 3D graphics cards implemented a fixed pipeline for the processing of primitive descriptions tuned as a state machine from an API such as OpenGL. But their previously fixed graphics pipeline stages were replaced with programmable components called *shaders* providing great versatility and power to the developer. The basic CPU/GPU architecture model is outlined in Figure 3a.

The hardware accelerated programmability of GPUs has been exposed to programmers for the

development of specialized programs called shaders. They are primarily used for rendering customized special effects and realistic 3D scenes in real-time. These shaders are loaded into the graphics card for replacing the fixed functionality. There are two kinds of shaders, respectively called vertex and fragment shaders. Originally they had to be coded in assembler, but as the graphics hardware increased in functionality and programmability, these shaders were more difficult to implement. Even more, the rapid evolution of GPUs forced to rewrite previous shaders to get maximum performance when a new family of graphics hardware were released. The solution to this low-level programmability came with the apparition of commercial high level shading languages and their compilers, which helped in portability and legibility, thus improving the development process.

In the context of computer graphics, a texture is an image that can be mapped to a polygonal structure to provide realism to the model. Basically, as an image, it can represent four values (R, G, B, A) as color and transparency components in every accesible location, called fragments or texels. The programmer is responsible for organizing the data in a grid to convert them into a texture, so creating textures in which texels keep numerical values of interest.

Textures are fixed to a well determined grid with the aim to operate on their texels. Then, a custom fragment shader is enabled and the operation kernel is executed over every fragment by simply rendering. An schematic view of this process is shown in Figure 3b. The programmability of the GPU at the fragment level is very well suited for stream computations. In its simplest form a kernel operation is executed over a large number of elements in a streaming single-instruction multiple-data (SIMD) fashion [11, 9].

The output result can be redirected to the input (by means of a pbuffer or some other alternative) in a multi-pass approach for continuing the processing task. At this point, it is important to remark that data readback from video memory to host memory is a well-known computational bottleneck. One of the hardest limitation of graphics hardware and its stream programming model is that there is no way to accumulate global computation values in the fragment combiner stage (i.e. no way to sum up all the values contained in a large texture in one rendering pass). The trick is to perform a so-called sum reduction where the size of the rendering target is halved in both directions making a 2x2 sum at the same rendering pass. Similarly, it is posible to perform a 1D sum-reduction halving the rendering target only in one direction. More information of this sum-reduction is available in [8]. This limitation leads to some implementation restrictions, such as we have to encode data in 2^nx2^m textures, filling with non-valid information the unnecessary space.

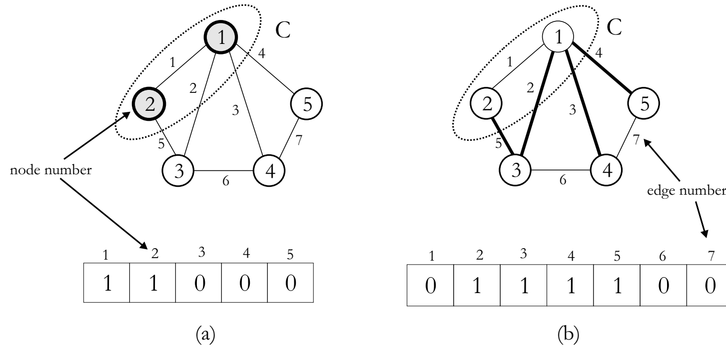

Figure 4: (a) Node-based representation of a posible Max-Cut solution. Nodes #1 and #2 are considered part of the solution. (b) Edge-based representation of a posible solution. Edges are lexicographically labeled and #2, #3, #4 and #5 are part of the considered solution.

4 Graph Representation

In this work, two representations of a Cut (C) are considered, and are shown in Figure 4. For explanation purposes we have only considered normalized graphs (edge weights equal to one). The node-based representation is shown on the left side. As the subgraph in C contains nodes #1 and #2, the solution is [1 1 0 0 0]. On the right side of Figure 4 edges that are cut by C are considered as part of the solution, so this representation is named as edge-based representation. In the example, edges shown in thick lines (#2, #3, #4 and #5) are considered as the solution which is represented by [0 1 1 1 1 0 0].

4.1 Node-based representation

Figure 5 shows the generation and evaluation of new solutions that belong to a given neighborhood. Lets suppose a starting point with a given solution [1 1 0 0 0] (Figure 5 a), we have to evaluate the movement in which node #1 is going out from subgraph C. The starting solution is repeated along rows of a nxn solution matrix (Figure 5 b). In another matrix, a negated transposed solution matrix is kept (Figure 5 c). On the other hand, two matrices are generated with the movement encoding. The first one has 1's in its first column because the considered movement is for node #1 (Figure 5 d). The second movement matrix is the first one transposed (Figure 5 e). These matrices are combined by means of logical operations (Figure 5 f, g and h) in order to create the access matrix to the adjacency matrix (Figure 5 i). From the element-by-element product a resulting matrix is obtained

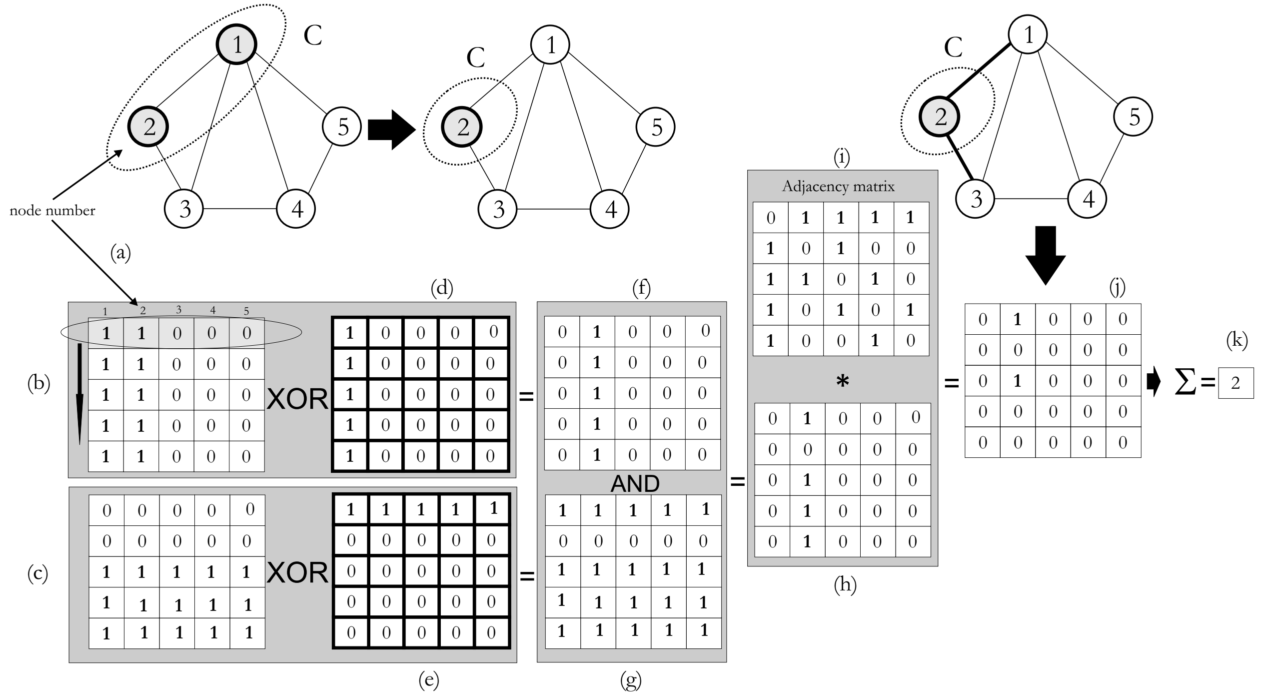

Figure 5: Matrix-based solution of the VNS applied to the Max-Cut problem. Thick line matrices are responsible of the movement.

in which survivor elements are edge weights of the Cut (C) after the movement, and the rest are 0 (Figure 5 j). Finally, the cost of the solution is determined by summing up all the elements of the matrix (Figure 5 k). Taking into account the architecture of the GPU, with this representation only 4 solutions are obtained in each rendering pass.

4.2 Edge-based representation

Edge-based representation is described by a 1xm-vector. Each position contains value 1 if the labeled edge is cut by the cutset C and 0 otherwise (Figure 6 a). The following two matrices corresponds to the n-repeated solution (Figure 6 b) and the cocycles matrix (Figure 6 c). An XOR operation is performed giving the neighborhood structure explained in Section 2 (Figure 6 d). The new Max-Cut value is obtained summing up each row of the resulting matrix and getting the maximum. This fact is equivalent to the inclusion or exclusion of a node in the considered solution (Figure 6 f).

This kind of representation is more adequate to the GPU computation model because data are more compact. In this case, each movement is described by means of each cocycle row. Taking

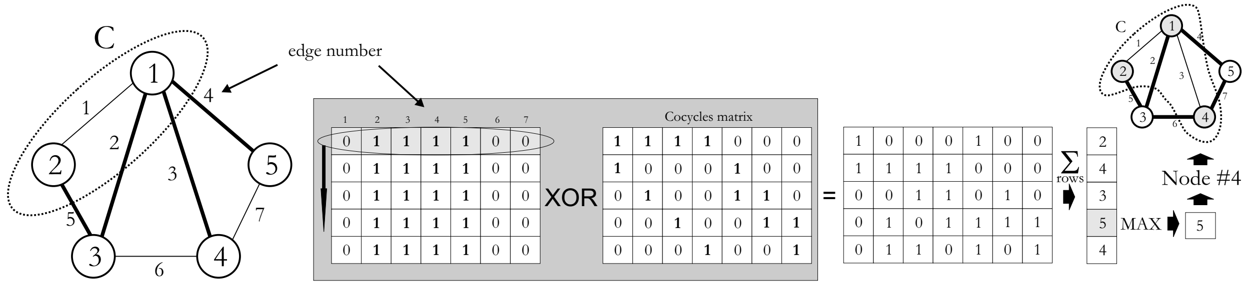

Figure 6: Edge-based solution of the VNS applied to the Max-Cut problem.

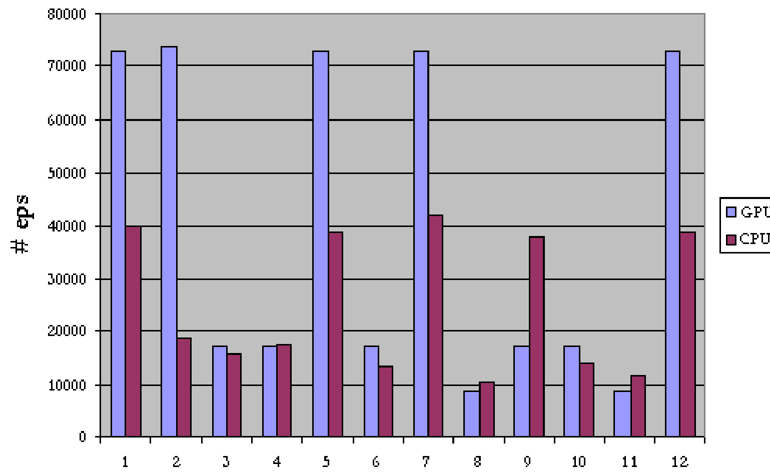

Figure 7: Comparison chart in number of evaluations per second in GPU and CPU.

into account the architecture of the GPU, with this representation $4n$ solutions are obtained in each rendering pass. Although this representation can be conceptually more difficult than the previous one, it gets n times more evaluations than the former.

5 Experimental Results

Experimental results were performed in a 2.7GHz Pentium 4, 1GB RAM and for the graphics platform an Nvidia GeForce 6800GT. For optimization purposes, we have only considered the edge-based representation of the graph. Table 1 shows information related to them (number of nodes, edges and their ratio) and number of evaluations per second (eps) for each platform.

	# Nodes	# Edges	E/N Ratio	#eps GPU	#eps CPU
1	800	1600	2	72704	39924
2	2000	4000	2	73728	18475
3	3000	6000	2	17408	15587
4	2250	10000	4.44	17408	17465
5	1000	3000	3	72704	38548
6	3000	9000	3	17408	13492
7	900	4000	4.44	72704	41892
8	4000	16000	4	8704	10483
9	1000	5000	5	17408	37884
10	3000	15000	5	17408	13919
11	3500	14000	4	8704	11699
12	1000	4000	4	72704	38575

Table 1: Results with graph configuration and number of evaluations per second (#eps)

In Figure 7 results are presented in a bar diagram. As it can be observed, the GPU gets much better throughput for the 66% of the benchmarks. The key factor consists of the edge-node ratio, and for a high one the CPU can outperform the GPU.

6 Conclusions

In this paper we have presented a high performance implementation of the VNS metaheuristic for solving the Max-Cut problem on sparse graphs. This problem has to be mapped to a matrix configuration in order to adapt data structures to a graphics processing unit (GPU). A node-based and edge-based representation are proposed, but in the experimental results we have noted that the second one gives better throughput. The parallel GPU approach can outperform the traditional CPU implementation in 66% of the considered benchmarks even quadrupling performance.

References

[1] S. Binato and G.C. Oliveira. *Essays and Surveys in Metaheuristic*, chapter GRASP: An Annotated Bibliography for Transmission Network Expansion Plannig, pages 325–367. Kluwer Academic Publishers, 2002.

[2] S. Burer, R. D. C. Monteiro, and X. Zhang. Rank-two relaxation heuristic for the max-cut and other binary quadratic programs. *SIAM Journal of Optimization*, 12:503–521, 2001.

[3] M. X. Goemans and D. P. Williams. Improved approximation algorithms for max-cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42:1115–1142, 1995.

[4] P. Hansen and N. Mladenović. Developments of variable neighborhood search. In C.G. Ribeiro and P. Hansen, editors, *Essays and surveys in Metaheuristics*, pages 415–439. Kluwer Academic Publishers, 2002.

[5] M. J. Harris. GPGPU. General Purpose Computing using GPUs website. http://www.gpgpu.org, 2002.

[6] M. J. Harris. *Real-Time Cloud Simulation and Rendering.* PhD thesis, University of North Carolina at Chapel Hill, 2003.

[7] R.M. Karp. *Reducibility among Combinatorial Problems*, chapter in Complexity of Computers Computation. Prenum Press, New York, USA, 1972.

[8] J. Kruger and R. Westermann. Linear algebra operators for gpu implementation of numerical algorithms. *ACM Transactions on Graphics*, 22(3):908–916, 2003.

[9] M. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Transactions on Graphics*, 23(3):787–795, 2004.

[10] K.-S. Oh and K. Jung. Gpu implementation of neural networks. *Pattern Recognition*, 37, 2004.

[11] S. Venkatasubramanian. The graphics card as a stream computer. In *SIGMOD-DIMACS Workshop on Management and Processing of Data Streams*, 2003.