Mapping Irregular Computation onto the Graphics Pipeline

Manuel Ujaldon [*]

Computer Architecture Department, University of Malaga, Spain

Joel Saltz

Biomedical Informatics Department, Ohio State University, U.S.A.

Abstract

The paper describes a set of strategies for mapping irregular codes onto commodity graphics hardware for its efficient execution. We start identifying the resources that current GPUs contain for solving indirect array accesses entirely on hardware, like vertices, textures and color tables. We then show how multiple indirections can be mapped onto the graphics pipeline, basically taking advantage of its streaming architecture for sequencing the indirections through subsequent pipeline stages. Our techniques are applied over typical irregular kernels like the sparse matrix-vector multiply and the Euler solver. Execution times on the GeForce FX and 6800 consistently outperform the Pentium 4 and Athlon 64 processors even by a 400% gain. We also analyze the impact that floating-point precision has over performance, by taking advantage of 16-bit floating-point color channel representation recently introduced by Nvidia in its GeForce 6 Series.

Key words: Computer Graphics, Graphics Processors (GPUs), General-Purpose GPU (GPGPU), Graphics data structures, methodology and techniques.

[*] Corresponding author. Address: Computer Architecture Dept. Complejo Tecnológico, Campus Teatinos. Boulevard Louis Pasteur, s/n. 29071 Malaga, Spain
Email addresses: `ujaldon@ac.uma.es` (Manuel Ujaldon), `saltz@bmi.osu.edu` (Joel Saltz).

1 Introduction

The programmability and performance of modern graphics processors (GPUs) has increased at such a rapid pace that they are already capable of outperforming CPUs in some compute intensive applications. Since the performance difference is expected to increase in the future [KDR*03,SIA03], researchers are adapting computationally intensive general-purpose algorithms to run on GPUs [GPGPU], sometimes with outstanding speed-up versus their CPU counterparts. Among them, we find volume segmentation (10-20 times faster) [SHN03], surfaces deformation (10-15x) [LKH*03], multigrid solvers (3x) [GWL*03], linear algebra (2x) [BFG*03,KW03,FSH04] and database operations [GLW*04,US05].

This work focuses on irregular problems, a broad superset of codes characterized by indirect array accessing (i.e., A[B[C[i]]]) which have proven to be particularly tough for high-performance: On CPUs, for the lack of locality in memory references; on multiprocessors, for the challenging data partitioning. We have found GPUs very compelling for dealing with such applications, since its streaming execution model reverses the bottleneck inherent to memory access: Data are the axis flowing through the graphics pipeline, and instructions are those who come to meet them, maximizing throughput with extraordinary performance and scalability.

Our preliminary studies in 2004 revealed that the graphics bus was the actual bottleneck for the GPU. We already showed how to overcome this by overlapping computation and communication using recent OpenGL extensions such as `NV_fence`, `NV_vertex_array_range` and `NV_pixel_data_range`, and later combining this with the enhanced features provided by PCI-Express. In 2005, major constraints that general-purpose computing faces using the GPU involves more to the data values. First, inaccuracy of the computation due to poor floating-point precision; second, clamping final results to the (0,1) range. These two issues were improved in the GeForce 6800 family from Nvidia, and both are explored here along with performance/cost ratio versus the CPU.

2 Using the GPU for general-purpose applications

In general, a GPU accepts an input stream (vertex attributes), transforms it through a sequence of kernels or *shaders* (vertex program, fragment program, texture operators), and returns an output stream (rasterized pixels), which is written into the frame buffer. Using GPUs for general-purpose computation entails disguising input data as vertex attributes, large data structures as textures, instructions as kernels, and final results as portions of video memory.

Graphics processing	Conventional programming	Graphics processing	Conventional programming
Texture memory	Arrays in main memory	Geometry (T & L)	N-ary arithmetic operators
List of vertices	Inner loop(s) of a computational block	Blending functions	Reduction operators
Rendering passes	Outer loop of a computational block	Clipping the scene	IF within the inner loop
Vertex indexing	First (inner) level of indirection	Active window	IF within intermediate loops
Textures lookup	Intermediate levels of indirection	Color index mask	IF within the outer loop
Color tables	Last (outer) level of indirection	Multipass rendering	Kernel programming

Table 1
The GPU abstraction basics for a general-purpose programmer. Restrictions apply depending on the features of a particular graphics architecture.

To overcome such limitations, we have to set aside the traditional programming paradigm and focus on the data flow (the stream). Each building block of a program constitutes a stream of vertices, whose geometry is defined according to existing loops and conditionals in the block for the kernels to compute only the desired elements. Multipass rendering executes the blocks sequentially, with the frame buffer and textures allocated in video memory to be used for communicating consecutive blocks.

Table 1 shows a list of GPU-CPU equivalencies extracted from our experience when implementing codes on the GPU. Overall, graphics units are used for different purposes they are intended to, and our goal is to identify those program elements leading to a performance gain on the GPU. Code excerpts that execute better on the CPU may remain there, and executeAsync() calls can be used to exploit task parallelism between the couple. This way, the GPU is no longer a rival for the GPU, but a co-processor reducing its workload.

3 Streaming over the GPU

Typical GPUs nowadays surpass 200 million transistors running around 400 MHz. They consist of two programmable processors (vertex and pixel shaders), enhanced to contain a chain of texture shaders, each accepting as texture coordinates the output of a previous shader. This transforms the GPU into an indirection engine which can solve entirely in hardware accesses a great number of nested indirect arrays: One for vertices lookup, dozens for textures lookup, and a final indexed access to the color table.

3.1 Data locality

In order for the streaming model to achieve a strong data locality, each element X passing through the stream has to be accompanied by those data required to compute such an instruction. To fulfill this, we consider a memory hierarchy which stores each data at the appropiate level

(1) Input scalars: They are defined as part of the common geometry for the graphics problem, being generated directly from hardware.
(2) Input vectors: We classify arrays into two types: (a) All vectors sharing a common access pattern are defined as vertex attributes, which guarantees they travel together along the stream and will always be available for computation at any stage of the graphics pipeline. (b) Those arrays having a different access pattern are stored on 1D/2D/3D textures, with the pattern being sent in the stream architecture as texture coordinates. This enforces data of these vectors to be accessed in the final stages of the pipeline.
(3) Output results: The frame buffer stores the values computed for each vertex as final results, just the same way vertices contribute to compose the graphics scene on screen. Other areas from video memory may be used as well, and the whole data sent back to main memory when necessary.

3.2 Operators available

The GPU architecture is devoted to those operators typically involved in graphics computing. Those of particular interests for our purposes are:

(1) On the vertex shader: Simple vector-vector and matrix-vector linear algebra operators, as required by translations/rotations in coordinates space.
(2) On the pixel shader: Product of up to four elements are directly performed on hardware between the RGBA components for the input color attribute and the corresponding output coming from a texture. All the product operators in our codes were carried out using such facilities.
(3) Blending functions: Accumulative sum/subtraction, min/max for all input vertices matching the same pixel on screen are performed over the frame buffer, also for RGBA components as specified in OpenGL 2.0.

When an algorithm is more demanding with operators, shaders can always be programmed to extend the GPU functionality and satisfy our needs.

4 Solving indirections on the GPU

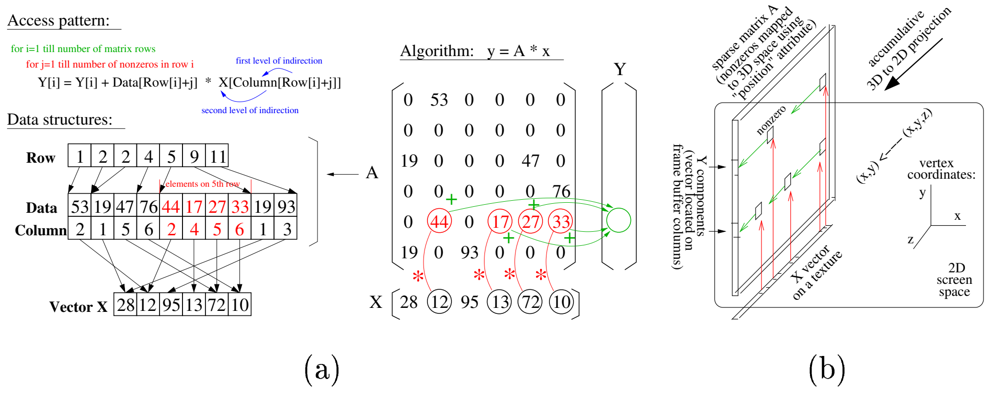
Irregular applications are mainly characterized for accessing data through indirections, that is, at least one array is being used as index to the one containing the actual value, namely Value(Index(I)). Codes containing indirections pose challenging issues for high-performance: On a single processor, they lack of data locality and cache memory is underused. On multiprocessors, it is difficult to partition code and data so that they match onto the same processor.

In contrast, we have found the GPU as a very appropiate platform for efficiently dealing with those elements. We solve indirect array accessing by transforming complex access patterns into series of direct references which flatten array acceses (the first indirection as vertex indices, intermediate indirections as texture coordinates, and even a final indirection as color indices). This way, all data are locally referenced and directly mapped onto the GPU hardware, without suffering any memory stall. As result, the code executes with a performance boost.

From now on, we focus on two clusters of problems typically dealing with indirections: linear algebra operators for sparse matrices and partial different equation solvers on unstructured meshes.

4.1 Sparse matrix algorithms: Indirections on the right hand side

The Sparse Matrix-Vector Multiply (SpMxV - see Figure 1.a) is a kernel used in a wide variety of iterative methods for solving linear systems [BBC*94], where indirections arise on data structures storing nonzeros in compressed formats like CRS (Compressed Row Storage), the method we use here for not imposing any restriction on nonzeros placement.



Access pattern:
for i=1 till number of matrix rows
for j=1 till number of nonzeros in row i
Y[i] = Y[i] + Data[Row[i]+j] * X[Column[Row[i]+j]]
first level of indirection
second level of indirection
Data structures:
Row
Data
Column
elements on 5th row
Vector X
A
Algorithm: y = A * x
Y
X
sparse matrix A (nonzeros mapped to 3D space using "position" attribute)
accumulative 3D to 2D projection
nonzero
Y components (vector located on frame buffer columns)
X vector on a texture
vertex coordinates:
(x,y,z)
(x,y)
2D screen space
(a)
(b)

Fig. 1. The Sparse Matrix Vector Multiply (SpMxV) algorithm, y = A*x, where A is stored in CRS format. (a) Access patterns and data structures. (b) The way we define the problem geometry to perform partial products using textures and a SUM reduction operator using blending functions on the frame buffer.

Figure 1.a shows how the matrix `A` is stored in CRS using three vectors: `Data`, with the nonzero values of the matrix in a row-major order, `Column`, for the column index for each nonzero, and `Row`, which marks the beginning of each row in the previous two vectors. The matrix can thus be traversed in our SpMxV code using two loops: The inner one iterating over the nonzero elements within a row, and the outer sweeping over the matrix rows. This way, `Row` and `Column` are both index arrays involved for composing access patterns with one or two levels of indirection, say `X[Column[Row[i]+j]]` for accessing a vector `X` to multiply with the j-th nonzero on i-th matrix row.

We mapped these vectors onto the GPU as follows:

X. We load `X` as a texture. 1D, 2D or 3D might be chosen, and we tried all three in our experiments, deciding in favour of a 2D texture, which was slightly faster (manufacturers claim they perform certain internal optimizations in 2D tectures, since these are by far the most common ones).

Y. We reserve a chunk of the frame buffer for the output vector, `Y`, where we will get the results at the end of the streaming execution (see Figure 1.b).

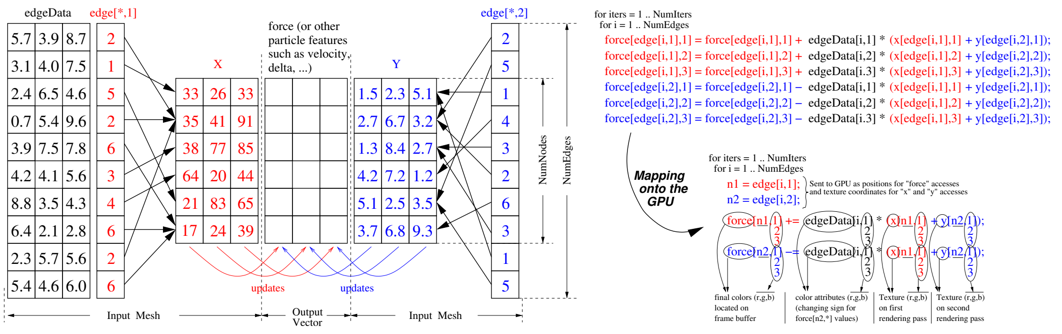
Data. Each nonzero is a vertex with the `Data` value in its color attribute to be multiplied by the `X` values accessed through (s,r) texture coordinates.

Row. It is implicit within the vertex locations defined to merge results onto the frame buffer. Let us initially assume that Y matches the first column of pixels on the screen. The vertex (nonzero) located on the matrix as the j-th element in i-th row then defines its position vector (x,y,z,w) as (0,i,j,0). If we set up a projection matrix so that the 3D vertex space be projected over the 2D image space, all vertices with the same `y` coordinate (that is, `i`) target the same pixel when rendered, located in screen coordinates (0,i) (see Figure 1.b). After passing the X texture stage, the actual vertex contribution is $Data * X$, and the blending function accumulates all different `j` contributions for the final result in the frame buffer to be $Y+ = Data * X$. When the size of `Y` is larger than a screen column, we just have to redefine the vertex positions to include additional columns in the `y` coordinates that where all zeroed in the previous scheme, till we fill the whole screen. Should the vector exceeds this size, we allocate it partially on a texture and program the pixel shader to act as an artificial blender. Note that geometry has implemented an access pattern in which `Row` is used as an indirection itself. In general, all indirections that are created with an index array being referenced in loop boundaries can be addressed this way.

Column. This is our actual indirection in the SpMxV problem, which is solved using three different methods: (1) **Vertex indices.** `Column` values are indices to the vertices whose texture coordinates point to the `X` texture. (2) **Texture lookup.** `Column` values are directly used as texture coordinates in the attributes for each vertex. This replicates coordinates for each vertex accessing the same `X` element, but benefits from a straightforward processing. (3) **Color table.** `X` values are placed into the color table, indexed by color indices travelling through the pipeline as `Column[j]`.

Section 5 discusses each of these methods from the experimental viewpoint when applied to the SpMxV.

Unstructured meshes provide a great deal of flexibility in discretizing complex domains and offer the possibility of an adaptive meshing. A typical example is the 3D Euler solver showed on Figure 2 [MAV91], which traverses the whole mesh calculating electrostatic forces between all pair of nodes connected through defined edges. The main differences with respect to the SpMxV lies in (1) the presence of indirections on the left hand side of assignments and (2) the vector type for the statements, which are replicated for the three components that each force (or velocity) has in the 3D space.



edgeData
edge[*,1]
force (or other particle features such as velocity, delta, ...)
X
Y
edge[*,2]
NumNodes
NumEdges
updates
updates
Input Mesh
Output Vector
Input Mesh
Mapping onto the GPU
Sent to GPU as positions for "force" accesses and texture coordinates for "x" and "y" accesses
final colors (r,g,b) located on frame buffer
color attributes (r,g,b) (changing sign for force[n2,*] values)
Texture (r,g,b) on first rendering pass
Texture (r,g,b) on second rendering pass

```
for iters = 1 .. NumIters
 for i = 1 .. NumEdges
  force[edge[i,1],1] = force[edge[i,1],1] +  edgeData[i,1] * (x[edge[i,1],1] + y[edge[i,2],1]);
  force[edge[i,1],2] = force[edge[i,1],2] +  edgeData[i,2] * (x[edge[i,1],2] + y[edge[i,2],2]);
  force[edge[i,1],3] = force[edge[i,1],3] +  edgeData[i.3] * (x[edge[i,1],3] + y[edge[i,2],3]);
  force[edge[i,2],1] = force[edge[i,2],1] −  edgeData[i,1] * (x[edge[i,1],1] + y[edge[i,2],1]);
  force[edge[i,2],2] = force[edge[i,2],2] −  edgeData[i,2] * (x[edge[i,1],2] + y[edge[i,2],2]);
  force[edge[i,2],3] = force[edge[i,2],3] −  edgeData[i.3] * (x[edge[i,1],3] + y[edge[i,2],3]);
```

```
for iters = 1 .. NumIters
 for i = 1 .. NumEdges
   n1 = edge[i,1];
   n2 = edge[i,2];
   force[n1,1] += edgeData[i,1] * (x[n1,1] + y[n2,1]);
            2                 2          2          2
            3                 3          3          3
   force[n2,1] −= edgeData[i,1] * (x[n1,1] + y[n2,1]);
            2                 2          2          2
            3                 3          3          3
```

Fig. 2. Data structures (left) and access patterns (right) for the 3D Euler kernel.

The average connectivity of each mesh node is between 6 and 10, hence the number of duplicate data references is low (input workload in Table 4 shows roughly five times more edges than nodes). This connectivity is particularly low when compared to other irregular problems such as molecular dynamics or particle dynamics. Once computed, the forces for each (i,j) edge are added to the total force associated with node i, and subtracted from the total force associated with node j. The main challenge this algorithm poses for high-performance computing is the data partitioning.

In distributed-memory multiprocessors, a block distribution for both data and indirection arrays results in a huge off-processor data, while a spectral bisection based on the connectivity of the mesh produces low volumes of communication between neighboring processors at the expense of a complex global to local index translation. Our GPU version does not require index calculation nor data fetching. All data are accessed locally taking advantage of the streaming model: The contribution of each i-node is entirely calculated on the first rendering pass, and the corresponding part for each j-node is added similarly on the second rendering pass. This straightforward strategy is possible because the loop does not contain any data dependencies. When dependencies arise, the GPU implementation is still possible, but requires to use additional elements of the graphics hardware and a more complex geometry definition for the problem.

Year	Central Processor (CPU)	Main Memory	Graphics Processor (GPU)	Video Memory
2003	Pentium 4 @ 2.4 GHz	1 Gb @ 4.2 Gb/s	GeForce FX 5900 @ 450 MHz	128 Mb @ 27.2 Gb/s
2004	Athlon 64 @ 2.0 GHz	2 Gb @ 6.3 Gb/s	GeForce FX 5950U @ 475 MHz	256 Mb @ 30.4 Gb/s
2005	Pentium 4 @ 3.2 GHz	1 Gb. @ 8.4 Gb/s.	GeForce 6800 GT @ 350 MHz	256 Mb @ 35.2 Gb/s

Table 2
Hardware features for our CPU-GPU comparison.

Sparse matrix	BCSSTK15	BCSSTK29	BCSSTK30
Rows	3948	13992	28924
Nonzeros	60882	316740	1036208
Fill rate	0.39 %	0.16 %	0.12 %
File Size (Kb)	1568	2680	8628
Vertex Buffer Size (Kb)	1894	9892	32378
Texture Size (Kb)	32	128	128

Table 3
Our input data set for the SpMxV algorithm. Matrices were taken from the Harwell-Boeing collection, where they are represented in CRS format.

5 Experimental results

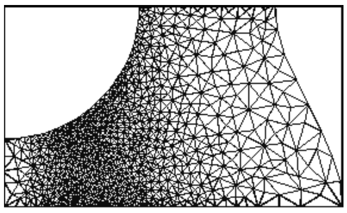
To demonstrate the effectiveness of our techniques, we have conducted a number of experiments on regular PCs. See Table 2 for hardware features. All GPUs belong to the GeForce family from Nvidia.

On the software side, we use OpenGL to map the graphics elements onto the GPU. Two recent OpenGL extensions were used for optimization purposes: `ARB_vertex_buffer_object` on the vertex shader (February 2003) allowed us an efficient use of video memory for vertices and colors, and `NV_texture_shader` (July, 2003) made it possible to tune the four texture shaders to our particular needs. For the codes on the CPU, we use Visual C++ 7.0 running under Windows XP. We disable vertical sync as well as antialiasing and anisotropic on the GeForces. OpenGL interleaved arrays were used for sending vertex attributes to the GPU, vertex positions were precisely calculated according to screen resolution to skip the interpolation phase, and `GL_POINTS` was selected as drawing primitive to keep computations strictly over the input list of vertices.

5.1 The Sparse Matrix-Vector Multiply

We run the SpMxV algorithm for three different sparse matrices coming from the Harwell-Boeing data set (see Table 3) [DGL92], roughly corresponding to a small, medium and large data set.

CPU versus GPU. Execution times are shown in Table 5. GPU time consistently outperforms CPU by roughly 50%.



(a)

Input mesh	2k	10k	50k
Nodes	2800	9428	53961
Edges	17367	59863	353476
File Size (Kb)	576	1810	11524
Vertex Buffer Size (Kb)	1085	3922	23163
Texture Size (Kb)	32	128	512

(b)

Table 4
(a) An unstructured mesh is composed of edges and nodes, with an average connectivity of about six. (b) The input data set used for our Euler solver.

Hardware Year	2003			2004			2005		
Input sparse matrix (BCSSTK#)	15	29	30	15	29	30	15	29	30
CPU time (msc.)	0.71	4.07	13.12	0.79	4.28	13.81	0.92	3.54	14.24
GPU time (msc.)	0.67	3.20	10.38	0.56	2.70	8.78	0.58	2.77	8.90
GPU Speed-up	1.05x	1.27x	1.26x	1.41	1.58	1.57	1.58	1.27	1.60
GPU Frames/sc.	1480	314	96	1785	370	113	1724	361	112
Total loading time	16.47	68.70	233.77	12.32	59.10	212.88	11.10	43.02	139.29
Iters. to amortize	499	78	85	53	37	42	32	55	26

Table 5
Performance numbers for the SpMxV running on the GPU under different sparse matrices.

Loading time. Loading time include the communication time for vertices and textures and also the task of filling the sending buffer to the GPU with the corresponding CPU data structures in an interleaved way. The size of this buffer is eight times the number of floating-point nonzeros, since we have (s,r) texture coordinates, (r,g,b) colors and (x,y,z) positions for each vertex. 2003 are raw numbers on the AGP bus. 2004 numbers were optimized overlapping communication with computation and using data prefetching (OpenGL extensions NV_vertex_array_range and NV_fence by Nvidia were needed - see [SE01]). 2005 loading numbers, benefit, in addition, from the PCI-Express enhanced features, reducing the I/O overhead roughly by 40%.

Indirection schemes. The `Column` indirection was implemented on the GPU using three different schemes: (1) Color table. Lookup time is almost negligible (0.01 msecs), since it is tightly coupled with the real-time constraints associated to the frame buffer. On the negative side, the fact the color table is right at the end of the graphics pipeline limits its application to the last indirection when nested. (2) Textures. Lookup spends 0.65 msecs. This is the most versatile scheme, hosting very large vectors and being capable of solving up to four nested indirections. In order to fairly compare GPU and CPU execution times, this has been the method selected in the GPU. (3) Vertex indexing. This is the slowest mechanism (2.40 msecs) barely useful for reducing the bandwidth requirements during the definition of geometries with strong data redundancy.

Type of memory. Before using the OpenGL extension `ARB_vertex_buffer_object`, we run our SpMxV code using main memory (DRAM) for allocating the GPU

Hardware Year	2003			2004			2005		
Input data mesh	2k	10k	50k	2k	10k	50k	2k	10k	50k
CPU time (msc.)	1.99	7.11	59.70	1.37	4.76	43.86	1.70	5.76	32.44
GPU time (msc.)	0.79	2.40	14.14	0.62	2.04	11.96	0.63	2.08	12.12
GPU Speed-up	2.51x	3.00x	4.25x	2.20x	2.33x	3.66x	2.69x	2.76x	2.67x
GPU Frames/sc.	1308	415	71	1612	490	83	1587	480	82
Total loading time	18.13	56.96	323.86	17.53	55.54	315.25	13.10	38.28	215.19
Iters. to amortize	15	12	7	23	20	9	12	10	10

Table 6
Performance numbers for the Euler kernel using several input meshes.

data structures. Execution times skyrocketed to four times those given in Table 5, suggesting that current GPUs are severely limited by bandwidth. Once data reach video memory, computation is faster than in the CPU.

Packaging strategies. For the SpMxV to benefit from GPU vector capabilities, some kind of data packaging can also be applied. One possibility is to use the OpenGL extension `ARB_multitexture`, loading four textures in such a way that X elements are placed on the R, G, B and A color component, respectively; then apply all four textures together (padding four nonzeros on a single color attribute), each with its own indexing, to give the result. Such computation is fast, but the loading time required for texture coordinates would hurt performance.

5.2 The Euler kernel

The input data set for the Euler solver consisted of three meshes obtained from real problems at NASA ICASE (see Table 4), again characterizing low, medium and high workload for computing. GPU improvement when compared to the CPU time is much higher than in the SpMxV case, with cases over 4x factor and an average gain of 270%.

Vector processing. We have found the major culprit for this outstanding results to be the GPU vector processing, which exploits the natural fact that every force in the Euler kernel possesses three components. The cost for computing four color components at a time within the GPU is just the same, but for the CPU the execution time multiplies by such 4x factor.

Loading time. It can be quickly amortized (barely 10 iterations in 2005) thanks to the low vertex attributes / actual operands ratio. A single (x,y,z) position and (s,r) texture coordinate is shared among the three components of the force, `X` and `Y` arrays, thus describing the geometry for the problem in a very compact manner.

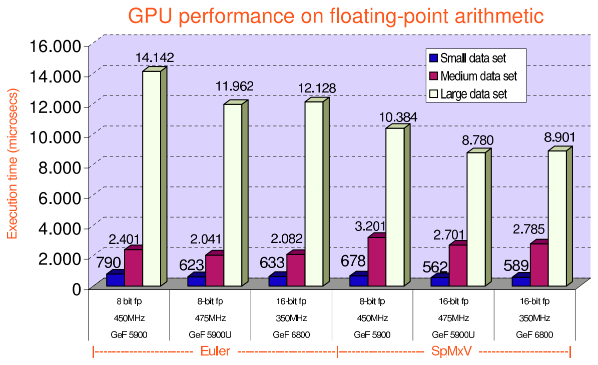


Fig. 3. Comparison in the GPU execution time depending on the floating-point arithmetic available in the final stages of the graphics pipeline.

6 Fidelity of the results

The use of GPUs for general-purpose computing is becoming increasingly popular with a wide number of examples outperforming CPUs [GPGPU]. Some of these comparisons have been criticized for being unrealistic and/or unfair due to clamping values and floating-point inaccuracy. Fortunately GPUs are evolving fast in resolving such shortcomings.

6.1 Clamping values

In the GeForce FX family, rasterization stage was clamping vertex attributes to the (0,1) range unless you use programmable shaders to bypass the stage. Later on, RGBA color was also clamped to (-1,1) or (0,1) depending on the data type to be declared as signed or unsigned, respectively. The GeForce 6 Series platform gave us the opportunity for overcoming those effects, since color components are no longer clamped as long as the data type is declared as FLOAT in OpenGL [KIL04].

6.2 Floating-point precision

After clamping colors in the GeForce FX models, a formula for converting the value into an internal data type was applied. Since 16-bit floating-point color representation was not available, results were converted into 8-bit integers per R,G,B,A component, which ended the computation with unrealistic values Texture values, on the other hand, are usually 8 bits long. 16 and 32 bits per color channel are provided under the `ATI_texture_float` OpenGL extension, and only certain drivers and hardware support it (in our equipment, only GeForce 6800 passed this test) The ATI Radeon family, uses a 24-bit floating-point representation for attributes and textures in its X300/600/700/800 models, as well as 16-bit floating-point representation per R,G,B color component.

Our first experiment concerning precision was to measure execution times using textures with 8, 16 and 32 bits textures per color channel. The GPU performance shown unaffected by this texture resolution, so accuracy may be improved in this stage without hurting efficiency.

Our second experiment enhances the floating-point precision using the High Dynamic Range 16 bit color OpenEXR representation available in our GeForce 6800. Execution times in Figure 3 (left for Euler, right for SpMxV) indicate that none of our codes was slowed down when ported to 16-bit floating-point precision. Even though we don't expect this feature to evolve on GPUs in the near term, performance on the GPU does not seem to be affected when increasing floating-point precision. Since we rely on blending functions for implementing reduction operators and the frame buffer for obtaining the results, our GPU implementation will not slow down in the future when 32-bit floating-point become available through the entire graphics pipeline.

7 Related Work

The first linear algebra implementation on a GPU was developed by Larsen et al [LM01] on a GeForce 3 (2001), where they run a dense 1024x1024 MxM in 546 msecs. Further improvementes followed [FSH04], but a limited number of efforts have targeted irregular computation. Two of the most recent cases came out simultaneously in SIGGRAPH'03:

Kruger et al. [KW03] implemented dense and banded matrices using textures and programming shaders. Execution times on banded matrices (no indirect addressing) with 10 diagonals of 4096 nonzeros were 0.72 msec., roughly the same we got with a sparse matrix (BCSSTK15) containing 50% more nonzeros without imposing any placement restriction. Also, their reduction operators required logN rendering passes using textures for storing intermediate results. Our reductions are accumulated on the frame buffer and performed on a single rendering pass, without worrying about the problem partitioning.

Bolz et al [BFG*03] solve the SpMxV on a GeForce FX storing all data on textures and programming the pixel shader with 33 instructions per matrix row, which enforces additional render passes every 200 nonzeros. They separate the main diagonal from the rest of the elements, and impose matrix rows to be sorted by the number of nonzeros per row, which is unrealistic in sparse applications. They perform 120 SpMxV operations per second over 37k vertices (8.33 msec. per SpMxV), 10 times slower than Kruger's for a similar workload.

Programming shaders force both approaches to decompose the problem using multirendering. This means computing the access indices at run-time, a major burden when indirections predominate. Where the access pattern remain constant through iterations, we set up the geometry to act as a tag for guiding the streaming computation, which extracts the entire index calculation out of the execution loops and amortizes the loading time through iterations.

Several general-purpose applications on GPUs dropped performance to levels comparable to CPUs when applied to 32-bit floating-point computations. In contrast, our codes consistently maintain improvement factors for both 8-bit and 16-bit color precision, given all remaining stages at 32-bit arithmetic. Benchmarking the NV30 fragment processor with simple vector operations revealed performance of 7 GFLOPS, which is 44% of its top performance. Tomov et al. [TMB03] also confirm this drop in their analysis, mainly due to bus and memory bandwidth, an issue recently improved using PCI-Express.

8 Conclusions

In this paper, we describe new methods for mapping irregular computation onto the graphics pipeline, showing how general-purpose applications can benefit from a streaming execution model to outperform current CPUs. Our methods avoid programming the shaders to overcome their current limitations so that the whole task can be performed on a single rendering pass.

Our work emphasizes the efficient resolution of indirect array accessing by transforming complex access patterns into series of direct or *flattened* references where all data are directly mapped into the GPU hardware. Average speed-up factor was 1.5x for SpMxV and 2.7x for Euler versus a CPU with five times higher frequency. Vector processing was proven to be very valuable and the graphics bus was identified as a bottleneck in the GPU. By overlapping communication with computation in 2004 and exploiting PCI-Express features in 2005 we were able to cut the I/O overhead almost by half.

Should we prioritize GPU hardware features for general computing according to performance/cost ratio, we enumerate: First graphics bus bandwidth, then video memory size and latency, and finally GPU frequency.

In the two year period we have based our survey, computed values were no longer clamped to the (0,1) range and color processing succeeded into a 16-bit floating-point representation for improving accuracy without hurting performance. Driven by the game industry, one can imagine GPUs continuing its fast evolutionary pace in functionality and performance so that virtually any application can be efficiently mapped onto the graphics pipeline.

References

[GPGPU] A Web page dedicated to the latest developments in general-purpose on the GPU. http://www.gpgpu.org.

[BBC*94] R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. var der Vorst. *Templates for the solution of linear systems: Building blocks for iterative methods.* Ed. SIAM, 1994.

[BFG*03] Bolz, J., Farmer, I., Grinspun, E., Schroder, P. *Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid.* Proceedings of the ACM 30th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'03), July, 2003. San Diego (California). Also available in ACM Transactions on Graphics, Vol. 22, n. 3, pages 917-924, 2003.

[DGL92] Duff I.S., Grimes R.G., and Lewis, J.G. *User's guide for the Harwell–Boeing sparse matrix collection (Release I).* Technical Report TR/PA/92/86, CERFACS, Toulouse, 1992.

[FSH04] K. Fatahalian, J. Sugerman, P. Hanrahan. *Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication.* Proceedings of the ACM SIGGRAPH - EUROGRAPHICS Workshop on Graphics Hardware (HWWS'04). Grenoble (France), August, 2004.

[GWL*03] Goodnight, N., Woolley, C., Lewin, G., Luebke, D., and Humphreys, G. *A multigrid solver for boundary value problems using programmable graphics hardware.* Proceedings of the ACM SIGGRAPH - EUROGRAPHICS Workshop on Graphics Hardware (HWWS'03). San Diego (California), pp.102-111, July, 2003.

[GLW*04] Govindaraju, N. K., Lloyd, B., Wang, W., Lin, M., Manocha, D. *Fast computation of database operations using graphics processors.* Proceedings 2004 ACM SIGMOD Int'l Conf. on Management of data, pages 215-226. Paris, France, 2004.

[KDR*03] Khailany, B., Dally, W., Rixner, S., Kapasi, U., Owens, J. and Towles, B. *Exploring the VLSI Scalability of Stream Processors.* Proceedings 9th Symposium on High Performance Computer Architecture. Anaheim (California), February, 2003, pp. 153-164.

[KIL04] Kilgard, M. *NVIDIA OpenGL Extension Specifications for the CineFX 3.0 Architecture (NV4x).* Nvidia Corporation, Mark J. Kilgard editor. May, 2004.

[KW03] Kruger, J., Westermann, R. *Linear Algebra Operators for GPU Implementation of Numerical Algorithms.* Proceedings of the ACM 30th Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'03), July, 2003. San Diego (California).

[LM01] Larsen, E. and McAllister, D. *Fast Matrix Multiplies using Graphics Hardware.* Proceedings Supercomputing 2001. Denver (Colorado), November, 2001.

[LKH*03] Lefohn, A., Kniss, J., Hansen, C., and Whitaker, R. *Interactive Deformation and Visualization of Level Set Surfaces Using Graphics Hardware.* Proceedings 14th IEEE Visualization Conference, Seattle (Washington), October, 2003, pp. 75-82.

[MAV91] D.J. Mavriplis. *Three dimensional multigrid for the Euler equations.* Journal AIAA, paper 91-1549CP, pages 824-832. June 1991.

[SIA03] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors.* Edition 2003.

[SHN03] Sherbondy, A., Houston, M., Napel, S. *Fast Volume Segmentation With Simultaneous Visualization Using Programmable Graphics Hardware.* Proceedings 14th IEEE Visualization Conference, Seattle (Washington), October, 2003.

[SE01] Spitzer, J. and Everitt, C. GL_NV_vertex_array_range and GL_NV_fence on GeForce Products and beyond. NVIDIA Corporation. August, 2001. "http://www.developer.nvidia.com/object/Using_GL_NV_fence.html".

[TMB03] S. Tomow, M. McGuigan, R. Bennett, G. Smith, J. Spiletic. *Benchmarking and Implementation of Probability-Based Simulations on Programmable GPUs.* Proceedings ACM Graphics Hardware Workshop, (W. Mark and A. Schilling editors). July, 2003. San Diego (California).

[US05] M. Ujaldon, J. Saltz. *The GPU as an indirection engine for a fast information retrieval.* Intl J. Electronic Business, 2005.