Experience with GPMAD

Sam Tygier

2013-1-16

Intro

- GPMAD is a tracking code that runs on a GPU
- It is designed to be similar to MAD
- It is written with CUDA

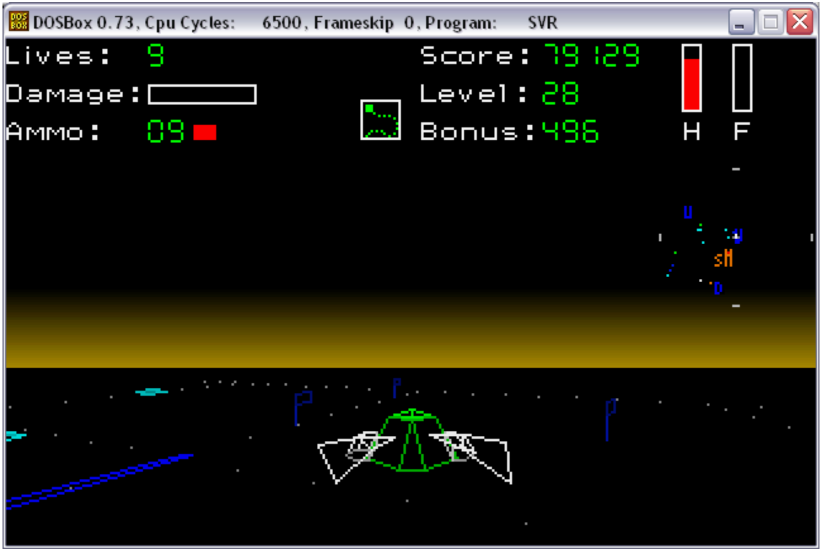
Talk will cover

- What is GP-GPU
- About GPMAD
- Experience of programming a GPU

Intro to GPUs

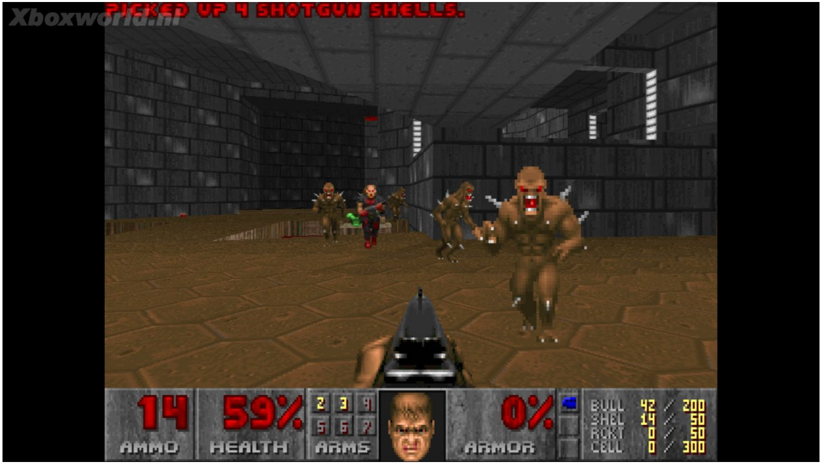
- Graphics card development has been driven by the computer game industry
- GPUs accelerator common operations to allow drawing of 3D scenes
  - Geometric transforms/projections
  - Transforming textures
  - Pixel effects (colour transforms, blur, anti-alias)
- Modern cards can draw billions of triangles per second

Game examples



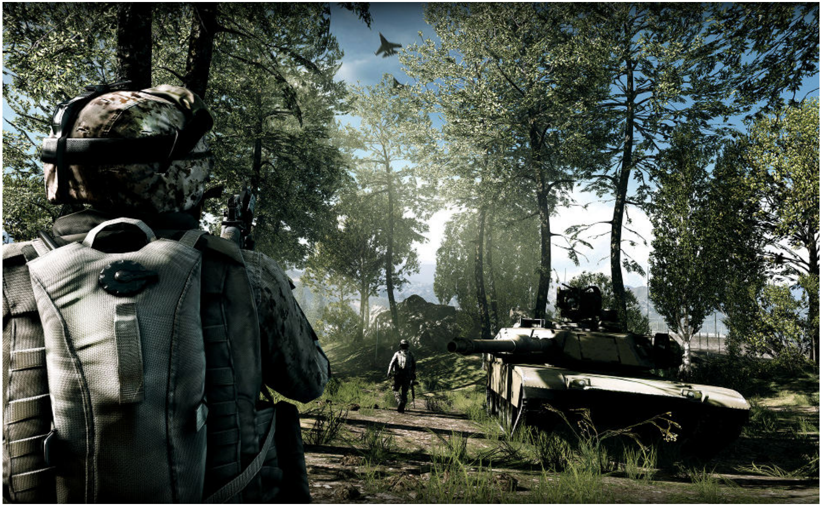
Spectre 1991

Game examples



Doom 1993

Game examples


Battlefield 2011

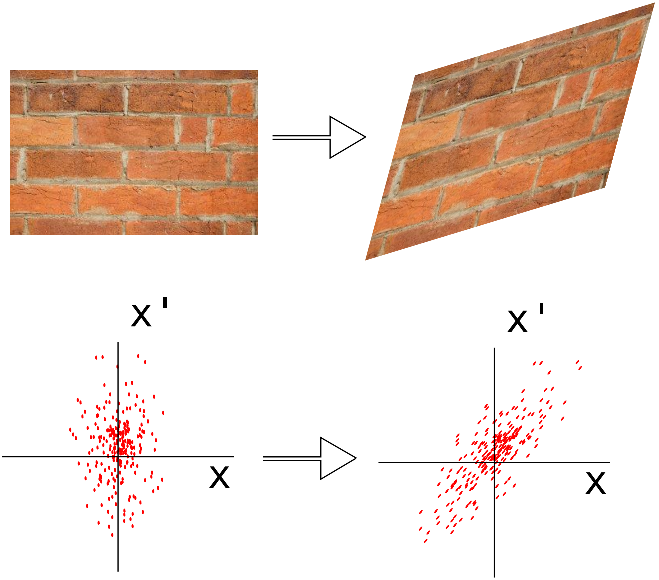
GP-GPU

- Early graphics cards where fixed function. But since around 2000 GPUs have offered programmable **shaders**.
- Initially separate **Vertex shaders** and **Pixel shaders**
- Todays cards have 100s of **Unified shaders**, analogous to CPU cores.

Lots of common algorithms are surprisingly similar to graphic operations. For example tracking a particle is very similar to rotating vertexes in 3D space.

- **GP-GPU** is General Purpose computing on a GPU

GP-GPU



Brook, CUDA and OpenCL

Brook

CUDA

OpenCL

- **Brook** was an early GP-GPU framework **Stanford University**
  - Converted work into something that looked like graphics work to the GPU
- **CUDA** is a proprietary framework from **Nvidia**
- **OpenCL** is an open framework from the **Khronos Group**
  - These bypass the graphics API and are far more suited for generic work

GPMAD

- GPMAD is a particle tracker inspired by **MAD**, which utilises GPUs.
- Has its origins in a MPhys project in manchester [1]
- J. Higham, M. Salt, R. Appleby, D. Bailey
- Originally used Brook
  - Hence limited by max texture size of the GPU (2048x2048) to 4 million particles
  - Limited to 16 bit floating point (paper says $<1\%$ errors after tracking a lattice)
- Found speedup of order 5 times compared to CPU (Nvidia 7900 vs Core2Duo?)
  - Limited by overhead below 10k particles

[1] High performance stream computing for particle beam transport simulations, R. Appleby, D. Bailey, J. Higham, M. Salt, CHEP07

GPMAD

- Since then GPMAD has been converted to **CUDA**
  - Number of particles only limited by GPU RAM (typically 1-4 GB)
  - Supports normal 32 and 64 bit floats
  - Simpler API (not a hack around a graphics API)
  - Nvidia only
- 100 - 500 times speed up (compared to MAD).
- Recently I have been working with Haroon Rafique to add space charge to GPMAD

GPMAD does not implement all of MAD. There would be little benefit for optics calculations or matching

GTX 460 vs Core 2 Duo

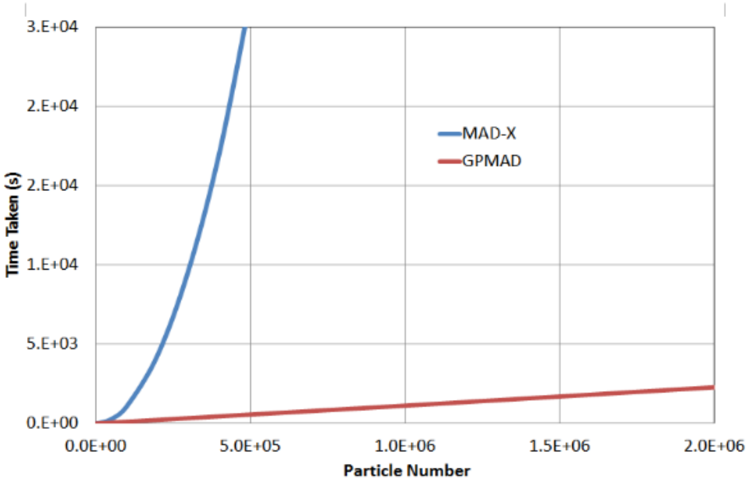


Figure 2: Run times for the DIAMOND BTS GPMAD compared to MAD-X (no space charge)

GPMAD basics

- GPMAD uses first or second order transport maps
- For each magnet in the lattice an *R* and *T* matrix are made. (Based on the MAD8 physics manual)
- Each particle is has 6 coordinates x, x_p, y, y_p, τ, p_t
- To track a bunch through an element each particle multiplied *R* and *T*

$$Z_j = \sum_{k=1}^{6} R_{jk} X_k + \sum_{k=1}^{6} \sum_{l=1}^{6} T_{jkl} X_k X_l$$

GPMAD usage

- GPMAD uses MAD-X compatible input
- Has a parser capable of interpreting complex lattices
- It then builds the list of matrices
- It then tracks the particles though the lattice using the GPU

GPMAD implementation

In the no space charge case:

- The lattice is uploaded to the GPU
- The particles are uploaded to the GPU
- The tracking kernel is run
- Twiss (Courant and Snyder) parameters at each step are stored in an array
- Finally output is copied back to main memory

GPMAD implementation

With space charge enabled.

- Input uploaded as before
- A kernel tracks through 1st half of a magnet
- Space charge kernel is run
- A kernel tracks through 2nd half of a magnet
- Repeat with second magnet

This prevents having an overly complex kernel

- Particles are kept on GPU whole time
- Kernel calls are asynchronous so must explicitly wait for them to complete

CUDA usage

Uploading the particles

```
particle* p_device_particle;
retval = (cudaMalloc(&p_device_particle, particlesize));
CUDACHECK(retval,"Malloc p_device_particle failed");
retval = (cudaMemcpy(p_device_particle, p_host_particle,
          particlesize, cudaMemcpyHostToDevice))
CUDACHECK(retval,"Memcpy p_device_particle failed");
```

- Allocate space on the GPU
- Copy data in
- Check that everything worked

CUDA usage

In the main code we call a wrapper

```
call_half_matrix_kernel(num_threads, num_blocks,
          p_device_particle, p_device_particle2, ...);
retval = cudaGetLastError();
CUDACHECK(retval," call_half_matrix_kernel");
```

This makes the actual kernel call

```
void call_half_matrix_kernel(...){
 dim3 grid( num_blocks, 1, 1); //blocks per grid
 dim3 threads(num_threads, 1, 1);//threads per block
 half_matrix_kernel<<<grid,threads>>>(...);
}
```

- A kernel is executed as a **grid** of **blocks** of **threads**

CUDA usage

In the main code we call a wrapper

```
__global__ void half_matrix_kernel(...){
 const size_t tid= blockIdx.x * blockDim.x + threadIdx.x;
 ...
 particle2[tid].x= map1[0][0] * particle[tid].x +
                    map1[1][0] * particle[tid].p_x + ...
 ...
```

Awkward bits

We often want to know statistical properties of a bunch

- Width
- Centroid
- RMS width
- RMS Twiss

Sum

```
double s = 0;
for(i=0;i<n;i++)
  s += part[i].x;
```

But seemingly simple operations like summing are non trivial in threaded code. You can't have 1000 threads simultaneously update the same memory location.

Awkward bits

- CUDA does not have a built in reduction operator like OpenMP.
- Can implement a tree based approach

$$\begin{tikzcd}
0 & 1 \arrow[dl] & 2 & 3 \arrow[dl] & 4 & 5 \arrow[dl] & 6 & 7 \arrow[dl] \\
1 & & 5 \arrow[dll] & & 9 & & 13 \arrow[dll] & \\
6 & & & & 22 \arrow[dllll] & & & \\
28 & & & & & & &
\end{tikzcd}$$

or

- Can take advantage of libraries that already exist

CuBLAS and Thrust
How ever these can’t be called from kernels, as they call their own kernels

GPU strengths

GPUs have some good strengths.

- High theoretical performance compared to CPU
  - FLOPS/£
  - FLOPS/watt

	Cores	GFLOPS single	GFLOPS Double	Power Watts	Cost
GTX 680	1536	3000	375*	200	£400
Radeon 7970	2048	3790	947	230	£400
Intel i7-3770	4	218	101	77	£250

*Limited to 1/8 of single. On Expensive Tesla cards 1/2 of single.

GPU weakness

- Harder to reach theoretical performance
- Requires manual memory management
  - Need explicit calls to copy memory to and from GPU
  - You need to minimise this to avoid performance hits
- Poor performance on branchy code
  - Cores within a group must be performing same operation. If threads follow different paths then other threads must wait
- Some tasks hard to parallelize
  - Hard to sum an array, and other operations useful for statistics
- Older GPUs were single precision only

Conclusions

- If you are currently tracking large numbers of particles with MAD have a look at GPMAD

`http://www.hep.manchester.ac.uk/gpmad/`

- If you are performing the same operation on large numbers of array elements have a look at GP-GPU
- However, beware that GPUs are limited in what they can perform. The bits that they can’t do will be your bottle neck
- If you are starting fresh then OpenCL allows your code to run on a wider range of hardware