Parallel Implementation of a Visual Attention Model

Anis RAHMAN

June 28, 2010

Table of contents

Overview

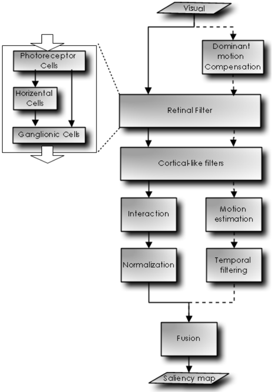
- GPUs and vision systems
- Case study: Visual attention model

Vision systems

- Sub-field of Computer vision
- To build artificial systems
- Involves large sets of data
- Same set of operations on it

Case study: Visual saliency model

- To find the spotlight of focus
- Based on human visual system
- Bottom-up model
- Implements both pathways
- Some applications:
  - To automate cinematography, surveillance, and video reframing
  - To simulate mediated reality
  - To find ROI maps



Saliency map

(a) Original image

(b) M_s

(c) M_d

(d) Saliency map

Characteristics of such applications

- Storage and memory usage
- Performance
    - Requires real time capability
    - Involves complex computations
- Data parallelism
    - Single operation on huge data
    - No or less dependency
- GPU friendly operations
- Previous attempts made required:
    - Learning graphics specific APIs
    - Re-structuring of algorithms according to the graphics pipeline

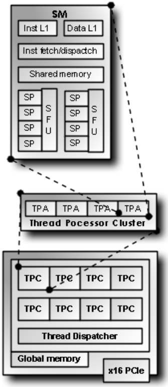
Why use GPU as co-processor?

- Stream processing architecture
  - All processors work in groups
  - Can communicate through shared memory
- Performance is unmatched
- Very high memory bandwidth
- Accessible
- Easier to program and manage
- Already applied in diverse fields
  - Biological engineering
  - Oil and gas exploration
  - Financial analysis

- To port data-parallel vision algorithm
- To demonstrate the speedup
- To confirm effects of low precision
- To apply different optimizations
- To experience the difficulties

The programming model

- CUDA(Compute Unified Device Architecture)
  - Independent of traditional graphics APIs
  - Based on:
    Arithmetic intensity = Arithmetic / Bandwidth
  - Uses familiar C
- Allows access to on-chip shared memory
- Provides texture lookups
- Supported by GPU-specific libraries
  - CUFFT, CUDPP, CuBLAS, OpenVIDIA



The programming model

- Code is composed of:
  - Host code
  - Kernel code
- Two-level thread hierarchy
  - 2D grid of thread blocks
  - Each thread block is 3D grid of threads
- Making the code scalable

How to program?

Restructuring the algorithm

- Identifying the data-parallel portions
- Avoiding code divergence

Effective use of memory model

- Constant cache for persistent data values
- Texture cache for frequently accessed data values
- Shared memory to reduce multiple accesses to global memory

Other options

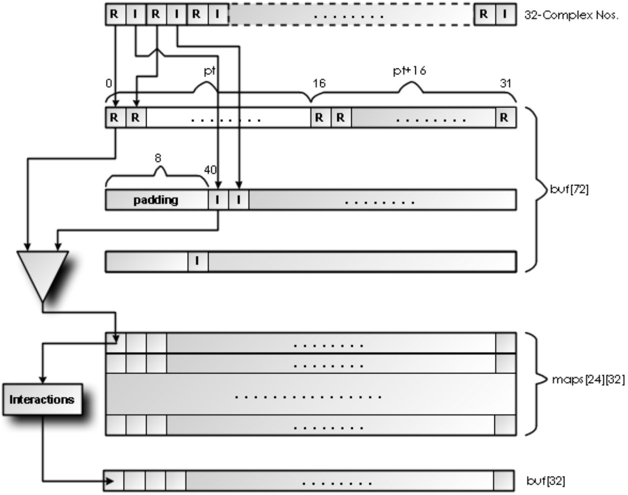
Using NVCC compiler options

- –use_fast_math: to use CUDA fast math functions
- –arch sm_13: to enable double precision (if supported)

Using GPU-specific libraries

- GPU-specific functions
- Black-box algorithm

Example: Interactions kernel



```
__global__ void ShortInteractionKernel ( Complex* in, unsigned int width
                   ,unsigned int height, float* out) {
  __shared__ float maps[ NO_OF_ORIENTS * NO_OF_BANDS ][32];
  __shared__ float buf [72];

  unsigned int x1 = blockIdx.x*blockDim.x + threadIdx.x/2;
  unsigned int x2 = blockIdx.x*blockDim.x + threadIdx.x;
  unsigned int y  = blockIdx.y*blockDim.y + threadIdx.y;

  if ( x1 >= width || x2 >= width || y>= height) return;

  unsigned int mod  = threadIdx.x1%2;
  unsigned int pt   = threadIdx.x1/2 + 40* mod;
  unsigned int size = width*height;

  for ( unsigned int j=0 ; j< NO_OF_ORIENTS ; ++j) {
    for ( unsigned int i=0 ; i< NO_OF_BANDS ; ++i) {
      /* ************************************************************
      * 32 threads process 16 complex numbers in parallel
      * every thread stores them with real and imaginary interlaced
      * 32 threads produce 32 real products in parallel
      ********************************************************** */
      buf[pt] = // first 16 complex numbers
        in[(j* NO_OF_BANDS+i)*size+(y*width+x1)][mod]/(float)(size);
      buf[pt+16] = // next 16 complex numbers
        in[(j*NO_OF_BANDS+i)*size+(y*width+x1+16)][mod]/(float)(size);
      __syncthreads();

      maps[j* NO_OF_BANDS + i][ threadIdx.x] = abs(
        buf[threadIdx.x ]*buf[threadIdx.x ] +
        buf[threadIdx.x + 40]* buf[threadIdx.x + 40]);
      __syncthreads ();
    }
  }
  // prefetched data in shared memory is used by interactions
}
```

Speedup

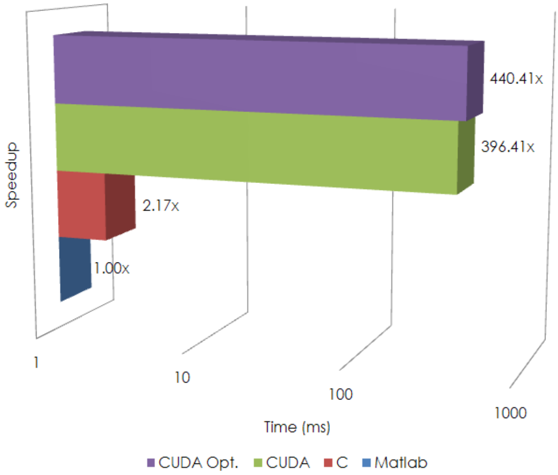


Figure: Speedup for the visual saliency model

Profile

Table: Computational cost of each step in static pathway

Kernel	Geforce GTX 260 (ms)	Geforce 8800 GTS (ms)
Mask	0.08	0.57
FFT	0.59	1.36
Shift	0.09	0.21
24×Gabor	1.47	6.31
24×Inverse shift	1.13	2.66
24×IFFT	10.76	32.54
24×Interaction	3.13	7.06
24×Normalize	3.33	25.27
24×Normalize Itti	3.34	25.37
24×Normalize Fusion	2.89	21.90
Total	26.81	123.24

Profile

Table: Computational cost of each step in dynamic pathway

Kernel	Geforce GTX 285 (ms)
Retinal Filtering	21.5
Modulation	2.6
Demodulation	3.1
Interpolation	0.3
Projection	0.2
Ver. Guassian recursive	33.2
Hor. Guassian recursive	21.7
Gradients	6.2
MCPI	39.1
Median filtering	0.2
Total	128.1

Precision

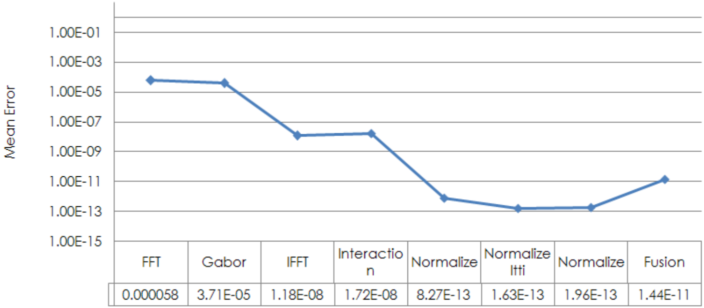


Figure: The effect of lower precision support on the result

Universal image index = Q = 99.66%

Profile

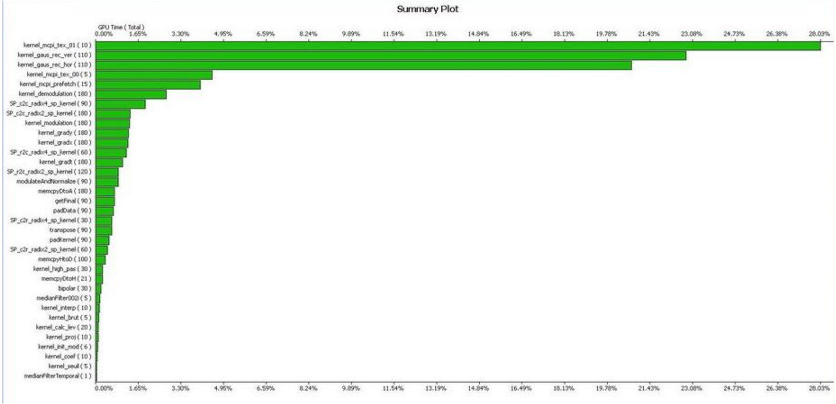


Figure: Profiling graph for the dynamic path

Multi-GPU solution

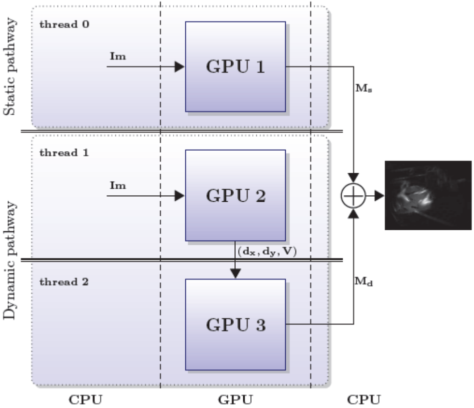


Figure: Block diagram of multi-GPU model

Issues

Memory size

- Only one image loaded to GPU, due to lack of GPU memory
- Causes frequent context switching
- CPU-GPU memory transfer i.e. one image per pass

Precision

- Doesnt fully conform to IEEE-754 standard
- Yet no full support for double precision
- Can lead in unusable results, due to lower accuracy
- Lower precision can be resolved using mixed precision

Conclusion

- Real time capability ($\sim$28 fps)
- Created opportunity to extend the model
- Exploited GPUs power without extensive re-structuring
- Without the need for high precision