CPU v GPU

A tale of three astro algorithms...

Jay Banyer

CAASTRO / Uni. of Sydney

Motivation

- The skeptical view:
  - GPU proponents show results for algorithms that are well-suited to GPU acceleration.
  - They also are implemented by experts, achieving the best results from the hardware.

- What about some real astro algorithms?
- What about the GPU beginner?
- How fast can *you* make *your* algorithms with a GPU?

Approach

- Three real astronomy algorithms:
  - Rotation Measure synthesis
  - Radio interferometer correlator X-engine
  - Radio image background RMS estimation
- Algorithms not chosen for suitability (or lack of) for GPU.
- Do the best CPU and GPU implementation I can.
- Compare results.
- Describe the optimisation process.

Disclaimer

- I am a beginner GPU programmer with only 3 weeks experience.
- The results are the best I could produce with limited time. More expertise and effort would likely produce (much?) better results on the GPU.

Overview

- Hardware used.
- For each algorithm:
  - Describe the algorithm.
  - Results.
  - Describe some optimisations used.
  - Discussion.
- Conclusions.

The Contenders...

Core i7-3770K ($330)
Core i7-2600K (EOL)

GT 640 ($130)
GTX 480 ($500)
GTX 690 ($1200)

Hardware Specs

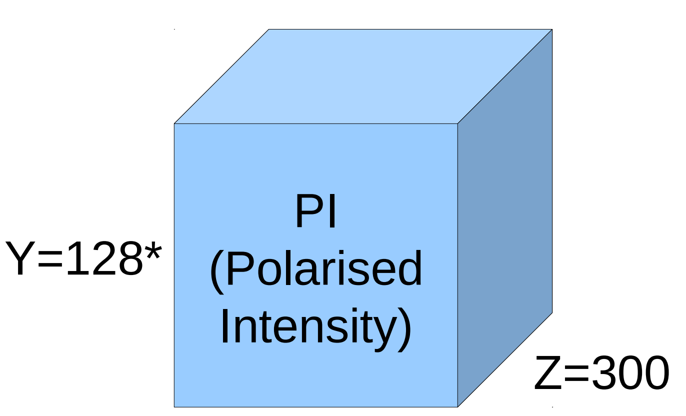
GPU Model	Arch.	Cores	RAM GB	RAM bandwidth GB/s	GFLOPS (FMA)	Street Price (AU)
GT 640	Kepler	384	2.0	28	690	$130
GTX 480	Fermi	480	1.5	177	1345	$500
GTX 690*	Kepler	2 x 1536	2 x 2.0	2 x 192	2 x 2810	$1200

* GTX 690 is a dual-GPU card. My test programs only used one chip using both requires explicit code.

CPU Model	Arch.	Cores	RAM GB	RAM bandwidth GB/s	GFLOPS	Street Price (AU)
i7-2600K	Sandy Bridge	4 (8)	8.0	2 x 12	~100?	EOL
i7-3770K	Ivy Bridge	4 (8)	?	2 x 12?	~100?	$330

Algo 1: Rotation Measure Synthesis

Inputs



F
(n=300)

Phi
(n=400)

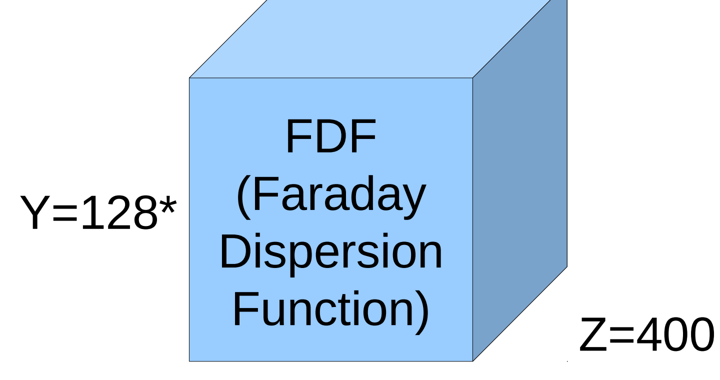
* Y=128 not realistic: limited to fit in GPU RAM

Processing

$$FDF(x,y,phi) = \sum_{k=1}^{300} PI_{x,y,k} * f(F_k, Phi_{phi})$$

```
for y in range(SIZE_Y):
  for x in range(SIZE_X):
    for p in range(SIZE_PHI):
      sum = 0+0j
      for f in range(SIZE_F):
        sum += PI[y][x][f] * func(F[f], Phi[p])
      FDF[y][x][p] = sum
```

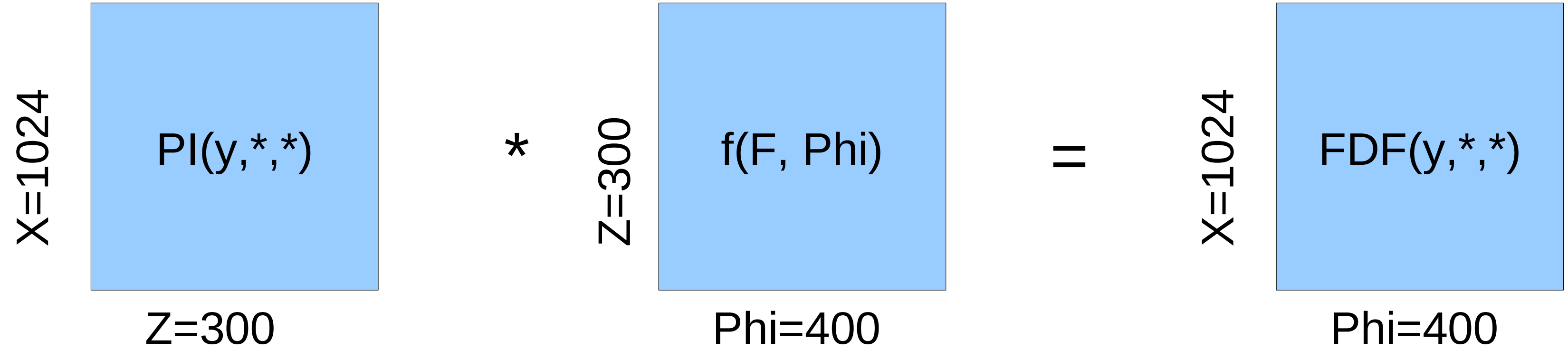
Output



Rotation Measure Synthesis

- Can be expressed as a series of 2D matrix multiplications.
- Allows use of BLAS linear algebra libraries, eg GSL, ATLAS, Intel MKL, CUBLAS

- For each y:



RM Synth (double) – all results

RM synth (double) - all results (lower is better)

cuda_by_hand_streams
cuda_cublas_streams
cuda_by_hand
cuda_cublas
mkl_cblas
atlas_ptcblas
carray_struct_openmp
atlas_cblas
carray_struct
vector_struct
gsl_cblas

GPU

CPU

0 5 10 15 20 25 30 35 40 45

Time (s)

GTX 690
GTX 480
GT 640
i7-2600K

RM Synth (single) – all results

RM synth (single) - all results (lower is better)

cuda_cublas_streams_sp

cuda_cublas_sp

mkl_cblas_sp

0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2

Time (s)

GTX 690
GTX 480
GT 640
i7-2600K

RM Synth – results summary

RM synth - fastest time (lower is better)

rm synth (single)

rm synth (double)

0 1 2 3 4 5 6 7 8

Time (s)

GTX 690
GTX 480
GT 640
i7-2600K

RM Synth – discussion

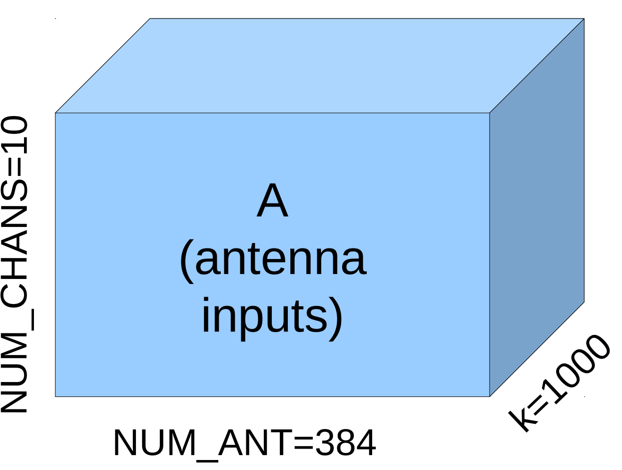
- Result: CPU and GPU similar speed.
- CPU
  - BLAS libraries (MKL, ATLAS) 2.2x faster than my hand-coded effort (for doubles). GSL is extremely slow.
- GPU
  - Limited by host<->device memory transfer. On GT 640 memcpy time is over 50% of compute time. Worse on faster cards.
  - CUBLAS library 1.4x faster than my hand-coded CUDA (for doubles). I expected a bigger difference.
  - GT 640 was unusually slow for double-precision. Why?
  - Streams didn't help much. Need to profile on faster cards to determine why.

GPU score card:

Algo 2: X-engine (based on SKAMP*)

Inputs

Per cycle:



Each value in A is a complex uint8.

Cycle is repeated 10 times.

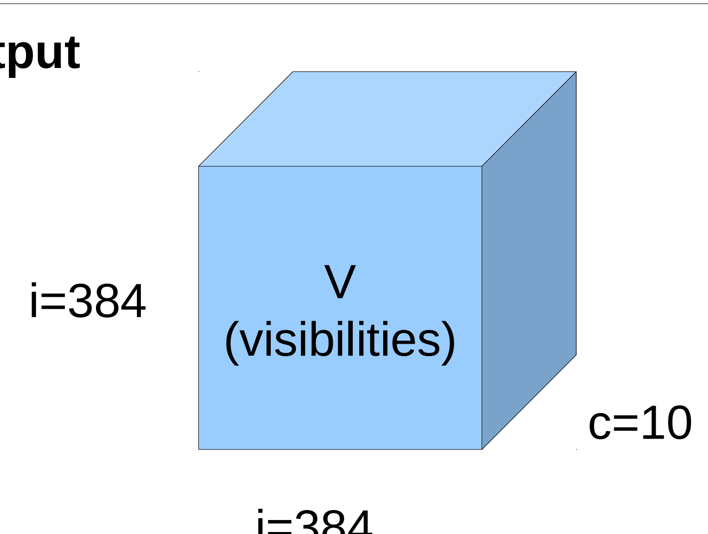
In the real system cycle is repeated 240 times before V is read out and reset.
And there are 12,480 channels :)

Processing

$$V(c,i,j)=\sum_{k=1}^{1000} A_{c,i,k} * \bar{A}_{c,j,k}$$

```
V[*][*][*] = 0+0j
for c in range(NUM_CHANS):
  for i in range(NUM_ANT):
    for j in range(NUM_ANT):
      for k in range(SIZE_STA):
        V[c][i][j] += A[c][i][k] * conj(A[c][j][k])
```

Output



Each value in V is a complex uint32.

* http://www.physics.usyd.edu.au/sifa/Main/SKAMP

X-engine – all results

xengine - all results (lower is better)

xengine_cuda1
xengine_cuda2
xengine_cuda3
xengine_cuda3s
xengine_cuda4
xengine_cuda4s
xengine_cuda5
xengine_cuda6
xengine1
xengine2
xengine3

0 10 20 30 40 50 60

Time (s)

GTX 690
GTX 480
GT 640
i7-3770K

multi-threaded

char->int (vectorisation)

X-engine – GPU optimisations

Optimisation: array layout

Slower:

```
__global__ void process_input(cmac_t* cmac, const input_t* i
{
  // which antenna pair (baseline) are we?
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  for (int k=0; k<STA_SIZE; k++) {
    cmac->vis[i][j].re += inp->samples[i][k].re*inp->samples[j][k].re - inp->samples[i][k].im*-1*
    cmac->vis[i][j].im += inp->samples[i][k].re*-1*inp->samples[j][k].im + inp->samples[i][k].im*
  }
}
```

Adjacent threads access memory STA_SIZE elements apart at each iteration. No coalesced loads :(

Guide: loop variable should be first index, thread index should be last (usually X). Opposite to CPU!

Faste

```
__global__ void process_input(cmac_t* cmac, const input_t* i
{
  // which antenna pair (baseline) are we?
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int i = blockIdx.y * blockDim.y + threadIdx.y;
  for (int k=0; k<STA_SIZE; k++) {
    cmac->vis[i][j].re += inp->samples[k][i].re*inp->samples[k][j].re - inp->samples[k][i].im*-1*
    cmac->vis[i][j].im += inp->samples[k][i].re*-1*inp->samples[k][j].im + inp->samples[k][i].im*
  }
}
```

Adjacent threads access adjacent memory elements at each iteration. Coalesced loads :)

Optimisation: shared memory

```
__global__ void process_input(cmac_t* cmac, const input_t* inp)
{
  // Use a block shared buffer to accumulate the 1000 inputs, rather than global memory
  __shared__ float buf_re[BLOCK_SIZE][BLOCK_SIZE];
  __shared__ float buf_im[BLOCK_SIZE][BLOCK_SIZE];

  // clear the shared memory buffer
  buf_re[threadIdx.y][threadIdx.x] = 0;
  buf_im[threadIdx.y][threadIdx.x] = 0;

  // which antenna pair (baseline) are we?
  int j = blockIdx.x * blockDim.x + threadIdx.x;
  int i = blockIdx.y * blockDim.y + threadIdx.y;

  for (int k=0; k<STA_SIZE; k++) {
    buf_re[threadIdx.y][threadIdx.x] += inp->re[k][i]*inp->re[k][j] - inp->im[k][i]*-1*inp->im[k]
    buf_im[threadIdx.y][threadIdx.x] += inp->re[k][i]*-1*inp->im[k][j] + inp->im[k][i]*inp->re[k]
  }

  // add the buffered results to global memory
  cmac->re[i][j] += buf_re[threadIdx.y][threadIdx.x];
  cmac->im[i][j] += buf_im[threadIdx.y][threadIdx.x];
}
```

Shared memory is much faster than global. Max 48KB per thread block.

1000 accumulations to shared memory instead of global.

Actually could just use local variables as there is no sharing between threads. Tried it – ran slightly slower. Perhaps due to more registers per thread.

X-engine – results summary

X-engine - fastest time (lower is better)

i7-3770K
GT 640
GTX 480
GTX 690

0 0.5 1 1.5 2 2.5 3 3.5 4 4.5

Time (s)

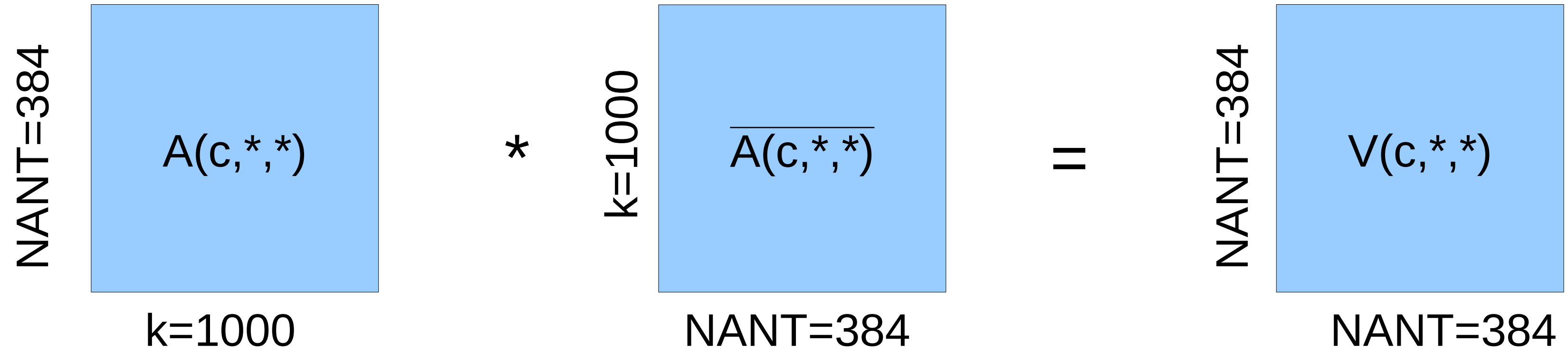
X-engine - discussion

- Result: GTX 480 6x faster than i7-3770K.
- CUDA streams didn't make a significant difference.
- GPU array layout in memory is critical. Ensure the *n*th thread accesses the *n*th element to allow coalescing.
- Using a shared memory buffer provided a big improvement.

GPU score card:

X-engine with Matrix Multiplication

- The X-engine can be implemented as a 2D matrix multiplication.
- For each cycle: for each channel c:



- We can use the BLAS libraries: **cblas_cgemm** does $C = \alpha*(A*B) + \beta*C$, where B is the conjugate transpose of A. Perfect!
- Requires changing from int to float as BLAS does not support int. This may not be okay in the real system.

X-engine MM – all results

X-engine – results summary

X-engine - results summary (lower is better)

xengine
xengine MM
0
0.5
1
1.5
2
2.5
3
3.5
4
4.5
Time (s)
GTX 690
GTX 480
GT 640
i7-2600K

X-engine MM - discussion

- GPU now only 2x faster than CPU.
- GT 640 faster than GTX 690! Something must be wrong.
- BLAS GPU versions bound by CPU to reorder input to column-major format (I think – not profiled). This can either be optimised or eliminated and GPUs would be faster.
- Result: I'm giving this one to the GPU.

GPU score card:

Algo 3: Background RMS

Image image:4

Flux distribution, RMS=0.422 mJy

Log Pixel Count

Flux (mJy)

RA (J2000)

Put a 64x64 box around each pixel.
Build a histogram of values in the box.
Determine the IQR from the histogram.
RMS at pixel = IQR / 1.34896

You can also determine IQR by copying and sorting each box but this is much slower than histogram.

Background RMS – all results

Background RMS - all results (lower is better)

	GTX 690	GTX 480	GT 640	i7-3770K
rms_cuda2s	~4.6	~13.4	~6.7	
rms_cuda2	~6.6	~8.9	~9.5	
rms_cuda1	~9.7	~8.0	~23.6	
rms_cpu1				~5.0

Time (s)

Background RMS - discussion

- GPU and CPU similar speeds.
- Quite a complex algorithm with lots of random memory access (populating the histogram bins). I was surprised the GPUs did okay.
- Result: not worthwhile on GPU.

GPU score card:

But wait, there's more!

- Let's try an algorithm that I think should be suited to GPU: 2D image convolution.

```
for y in range(IMAGE_SIZE):
  yoff = y-KERNEL_SIZE/2
  for x in range(IMAGE_SIZE):
    xoff = x-KERNEL_SIZE/2
    sum = 0
    for i in range(KERNEL_SIZE):
      for j in range(KERNEL_SIZE):
        sum += in[yoff+i][xoff+j] * kernel[i][j]
    out[y][x] = sum
```

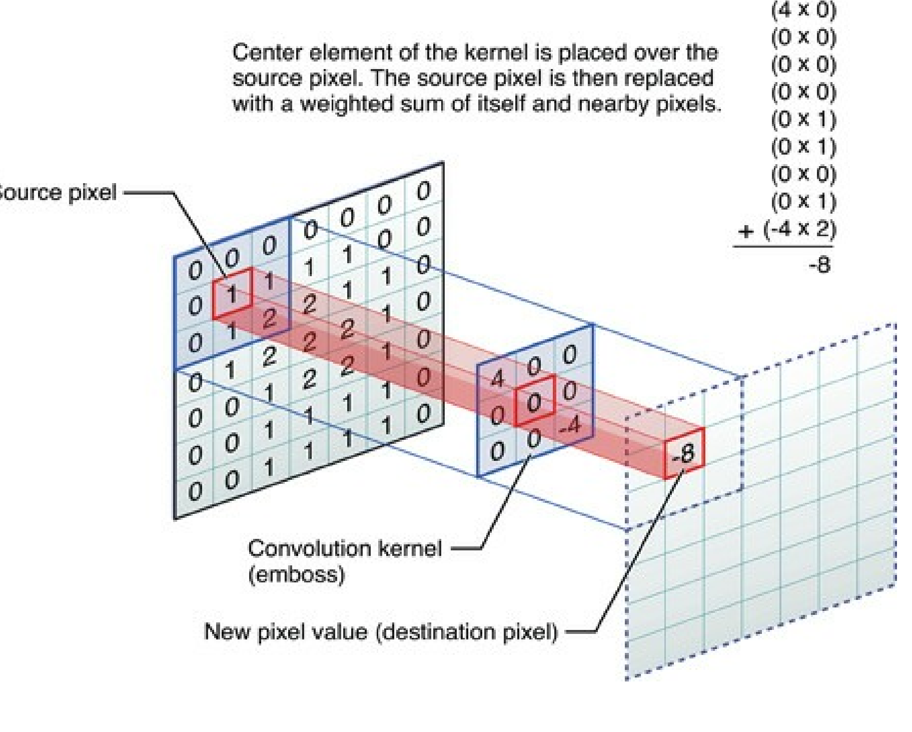


Image: developer.apple.com

http://en.wikipedia.org

Convolve

Convolve - all results (lower is better)

Optimisation: constant memory

```
// the convolution kernel goes in constant memory
__constant__ float c_kernel[KERNEL_SIZE][KERNEL_SIZE];

__global__ void convkern_pad(float* out, const float* inp_pad, size_t inp_pad_pitch,
int tilex, int tiley) {
  // index of pixel in image
  int x = blockIdx.x * blockDim.x + threadIdx.x + tilex;
  int y = blockIdx.y * blockDim.y + threadIdx.y + tiley;
  __shared__ float imbuf[IMAGE_CACHE_SIZE][IMAGE_CACHE_SIZE];
  const char* pinp = (const char*)inp_pad;
  int iy, ix;

  // cache the image data in shared memory.
  // each thread loads 9 pixels, 8 of which are in the surrounding apron
  for (iy=0; iy<IMAGE_CACHE_SIZE; iy+=BORDER_SIZE) {
    const float* inp_row = (const float*)(pinp+((y+iy)*inp_pad_pitch));
    for (ix=0; ix<IMAGE_CACHE_SIZE; ix+=BORDER_SIZE) {
      imbuf[iy+threadIdx.y][ix+threadIdx.x] = inp_row[x+ix];
    }
  }

  // apply the kernel
  if (x<IMAGE_SIZE && y<IMAGE_SIZE) {
    float v = 0;
      for (iy=0; iy<KERNEL_SIZE; iy++) {
        for (ix=0; ix<KERNEL_SIZE; ix++) {
          v += imbuf[iy+threadIdx.y][ix+threadIdx.x] * c_kernel[iy][ix];
        }
      }
    }
    out[y*IMAGE_SIZE+x] = v;
  }
}

// copy kernel data to GPU constant memory
cudaMemcpyToSymbol(c_kernel, kernel, kernel_bytes);
```

Declaration

Use as global

Assign using cudaMemcpyToSymbol

Convolve - discussion

- A simple algorithm, yet it required some effort to get good speed on GPU.
- Expert example at http://developer.download.nvidia.com/assets/cuda/files/convolutionSeparable.pdf
- Result: GPU 7x faster than CPU.

GPU score card:

Summary of results

Summary - fastest time (normalised to CPU, lower is better)

Time (normalised to CPU time)

3
2.5
2
1.5
1
0.5
0

rm synth (dp)
rm synth (sp)
xengine
xengine MM
rms
convolve

GTX 690
GTX 480
GT 640
Core i7

Final Points

- The GTX 480 (1345 GFLOPS, $500) performed better than the GTX 690 (2810 GFLOPS, $1200) in most cases. This is unexpected and may indicate a problem somewhere.
- Implementation and optimisation effort was similar for CPU and GPU: about 2-3 days per algorithm.
- How much faster could an expert make the GPU versions go? I suspect these GPUs could do much better.
- Optimisations I didn't try:
  - Texture memory.
  - Resource usage: registers per thread, L1/shared config.
  - Bank conflicts.
  - Others?

Tip: PCIe link width

NVIDIA X Server Settings

- X Server Information
- X Server Display Configuration
- X Screen 0
  - X Server Color Correction
  - X Server XVideo Settings
  - Cursor Shadow
  - OpenGL Settings
  - OpenGL/GLX Information
  - Antialiasing Settings
- GPU 0 - (GeForce GT 640)
  - Thermal Settings
  - PowerMizer
  - DFP-0 - (LG Electronics W2353)
- nvidia-settings Configuration

NVIDIA

PowerMizer Information

Adaptive Clocking:	Enabled
Graphics Clock:	901 Mhz
Memory Clock:	891 Mhz
Power Source:	AC
Current PCIe Link Width:	x1
Current PCIe Link Speed:	5.0 GT/s
Performance Level:	1

Performance Levels

Performance Level	Graphics Clock	Memory Clock
0	324 MHz	324 MHz
1	901 MHz	891 MHz

PowerMizer Settings

Preferred Mode: Prefer Maximum Performance

Help Quit

Should be x16!
x1 = 500MB/s
x16 = 8GB/s
Hardware issue
in my case, caused
slow memcpy.
Testing done at x8

Tool: Nsight (Eclipse)

C/C++ - xengine/xengine_cuda2.cu - Nsight

File Edit Source Refactor Navigate Search Project Run Window Help

Profile

Project Explo

- background_rms
- convolve
  - Binaries
  - conv_cpu_sp1.cc
  - conv_cpu_sp2.cc
  - conv_cuda_sp1.cu
  - conv_cuda_sp2.cu
  - conv_cuda_sp3.cu
  - conv_cuda_sp4.cu
  - conv.h
  - conv_cpu_sp1 - [x86_64
  - conv_cpu_sp2 - [x86_64
  - conv_cuda_sp1 - [x86_
  - conv_cuda_sp2 - [x86_
  - conv_cuda_sp3 - [x86_
  - conv_cuda_sp3s - [x86_
  - Makefile
- rm_synth
- xengine
  - Binaries
  - Includes
  - xengine_cublas1.cu
  - xengine_cublas2.cu
  - xengine_cuda1.cu
  - xengine_cuda2.cu
  - xengine_cuda3.cu

xengine_cuda4s.cu | xengine_cuda1.cu | xengine_cuda2.cu | »17

```
            }
        }
}

// process one STA (1000 samples) for each antenna for one channel.
// This performs the complex i*j and accumulation for each antenna pair.
// Each thread processes all 1000 samples for one antenna pair (i,j)
__global__ void process_input(cmac_t* cmac, const input_t* inp)
{
    // which antenna pair (baseline) are we?
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    for (int k=0; k<STA_SIZE; k++) {
        cmac->vis[i][j].re += inp->samples[k][i].re*inp->samples[k][j].re - inp->sam
        cmac->vis[i][j].im += inp->samples[k][i].re*-1*inp->samples[k][j].im + inp->
    }
}

void die_cuda(const char* msg) {
    fprintf(stderr, "CUDA error [%s]: %s\n", msg, cudaGetErrorString(cudaGetLastErro
    exit(-1);
}

int main(int argc, const char** argv) {
    int i, j, k;
    input_t* inp = new input_t[NCHAN];
    cmac_t* cmacs = new cmac_t[NCHAN];
    input_t* inp_d;
    cmac_t* cmacs_d;
    cudaDeviceReset();
    printf("Processing NLTA=%d NCHAN=%d STA_PER_LTA=%d\n", NLTA, NCHAN, STA_PER_LTA)
    printf("Per channel sizeof(input)=%lu sizeof(cmac)=%lu\n", sizeof(input_t), size
```

O

- stdio.h
- stdlib.h
- string.h
- cuda.h
- xengine.h
- BLOCK_SIZE :
- (anonymous)
- sample_t : str
- (anonymous)
- input_t : struc
- (anonymous)
- visibility_t : st
- (anonymous)
- cmac_t : struc
- generate_inpu
- process_input
- die_cuda(cons
- main(int, cons

Problems | Tasks | Console | Properties

No consoles to display at this time.

Writable | Smart Insert | 51 : 2

Tool: NVIDIA Visual Profiler

NVIDIA Visual Profiler

File View Run Help

*New Session | *New Session | *New Session | xengine_cublas2.cu

0.6 s | 0.65 s | 0.7 s

Process 23639
Thread 1088993088
Runtime API
Driver API
Profiling Overhead
[0] GeForce GT 640
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
99.9% [200] void ...
0.1% [1] memset (0)
Streams
Stream 1
Stream 10
Stream 11
Stream 12
Stream 13
Stream 14
Stream 15
Stream 16
Stream 17
Stream 18
Stream 19

Properties | Detail Grap

void magma_lds64_cgemm_kernel<boo...

Name	Va
Start	
End	
Duration	
Grid Size	
Block Size	
Registers/Thread	
Shared Memory/Block	
Occupancy	
Achieved	
Theoretical	
L1 Cache Configuration	
Shared Memory Requested	
Shared Memory Executed	

Analysis | Details | Console | Settings

Scope
- Analyze Entire Application
- Analyze Kernel (select in timeline)

Stages

Results

Low Memcpy Throughput [2.53 GB/s avg, for memcpys accounting for 100% of all memcpy time]
The memory copies are not fully using the available host to device bandwidth. More...

Conclusions and Advice

- GPUs are worthwhile for some algorithms. I suspect a small subset of real-world problems.
- Ask a GPU expert whether your problem is suited to GPU before trying unless you just want to experiment.
- CUDA tools (eg Nsight IDE, Visual Profiler) and documentation is good.
- CUDA programming is not too difficult although optimisation requires effort and knowledge, I think more so than CPU programming (for simple algorithms). You need to think differently and consider many factors.
- Be a good CPU programmer before trying the GPU.
- Your first GPU implementation can probably be optimised a lot – don't be put off by poor results from your first effort.