CUDA as a Supporting Technology for Next-Generation AR Applications

Tutorial 4

SIBGRAPI 2008

XXI BRAZILIAN SYMPOSIUM ON COMPUTER GRAPHICS AND IMAGE PROCESSING

CAMPO GRANDE/MS - BRAZIL

October 12-15, 2008

GRVM

CUDA as a Supporting Technology for Next-Generation AR Applications

Thiago Farias, João Marcelo Teixeira, Pedro Leite, Gabriel Almeida, Veronica Teichrieb, Judith Kelner
{tsmcf, jmnxt, pjsl, gfa, vt, jk}@cin.ufpe.br

Virtual Reality and Multimedia Research Group
Federal University of Pernambuco, Computer Science Center

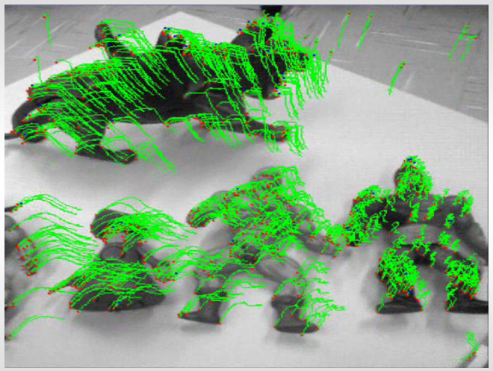
9/19/2008

Agenda

- Motivation
- CUDA Architecture
- System Configuration
- CUDA Programming Approach
- CUDA Programming Guidelines
- Case Studies
- Final Considerations

Motivation

Motivation



Resolution: 1024x1024
Number of features: 1000
Frames per second: ~50

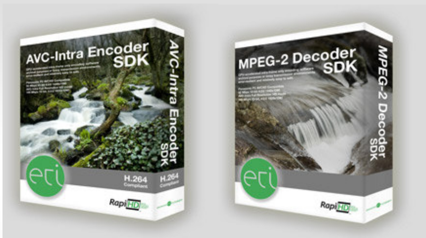
Motivation

- Computational power growth is (now) not sustained by processor clock
  - Stuck at ~3GHz by 2008
- Multi-core processing is the “new” way to increase speed
- Good coupling between some applications needs and the type of processing provided by the GPU

Motivation

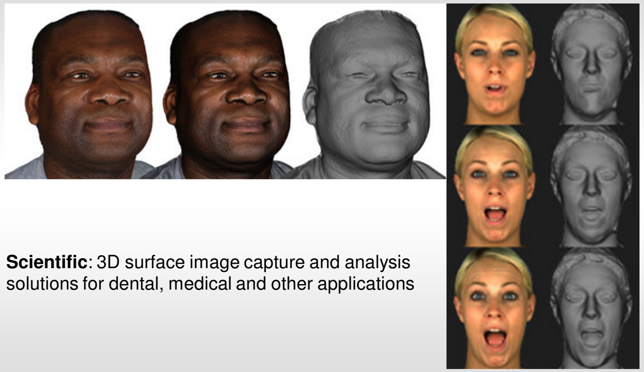
Games and simulation: GPU Physics

Motivation



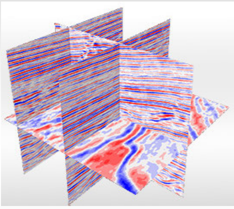
Video: GPU for High Definition encoding/decoding (h264/MPEG-2: RapiHD)

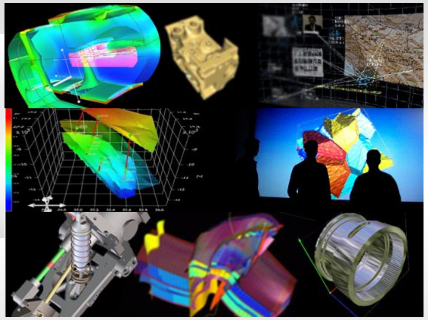
Motivation



Scientific: 3D surface image capture and analysis solutions for dental, medical and other applications

Motivation



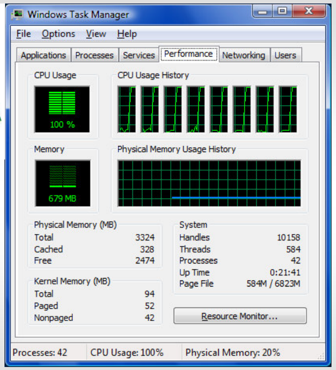


Oil & Gas/Energy/Engineering: Volumetric reconstruction, analysis and visualization

Motivation

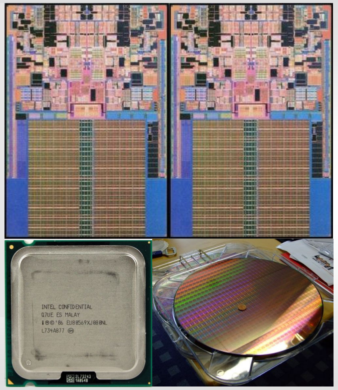
Augmented Reality: HD Video flows are now feasible

Motivation



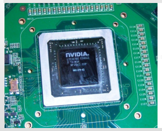
Motivation

	Core 2 Extreme QX9770
Clock Speed	3.20 GHz
FSB	400 MHz x 4
Cores	4
Cache	2x 6 MB
Process	45 nm
Transistors	820 million
Die-Area	214 mm²
TDP	136 W
Float OPs	42-57 GFLOPS
Price	~ US$ 1400.00

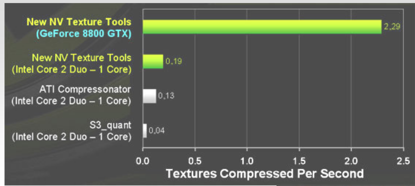


Motivation

	GeForce GTX280 (GT200)
Clock Speed	600 MHz (Shader: 1.30 GHz)
SPs	240
Memory	1024 MB
Process	65 nm
Transistors	1400 million (chip)
Die-Area	575 mm²
TDP	50-178 W (Card)
Float OPs	Up to 933 GFLOPS
Price	~ US$ 450.00

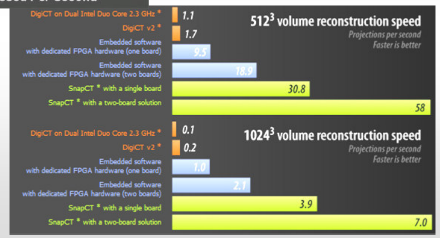


Motivation

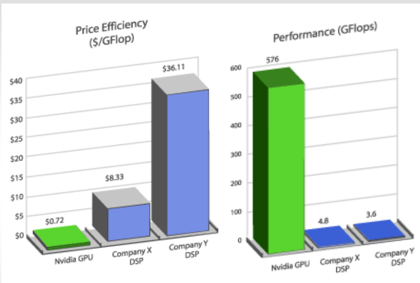


NVIDIA Texture Tools 2

SnapCT: tomographic reconstruction software



Motivation



RapiHD:

- More performance than the best DSP can offer.
- GPUs offer more performance per dollar.
- A single GPU has the computational power of more than 100 DSPs.

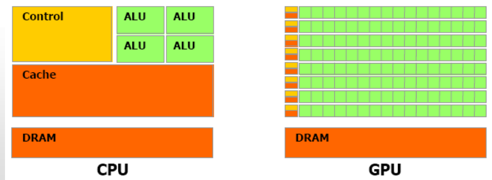
CUDA Architecture

CUDA Architecture

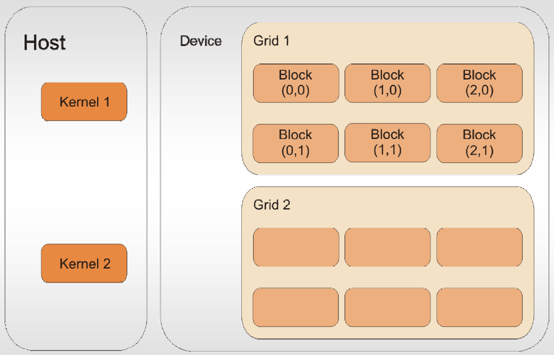
- CUDA requires Hardware-related knowledge
- To learn a different paradigm, the bottom-up approach is a good start
- PTX ISA
- Single and double precision
  - Speed, rather than precision, is a major concern when rendering 3D scenes, the main purpose of GPUs

CUDA Architecture

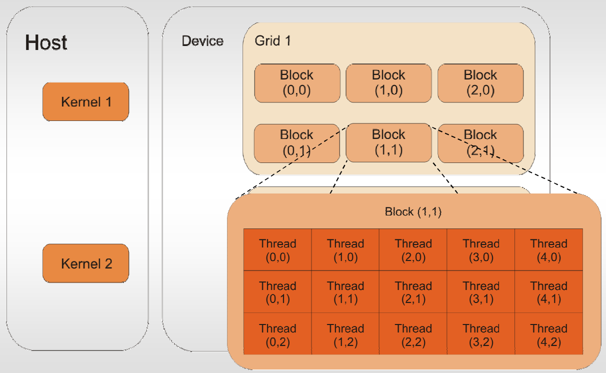
8800GTX (G80):
16 groups of 8 Scalar Processors (SPs) each, totalizing 128 processors. The groups, called multiprocessors, process blocks of 64 to 512 threads. Each multiprocessor contains 8 SPs and 2 Special Function Units (SFUs).

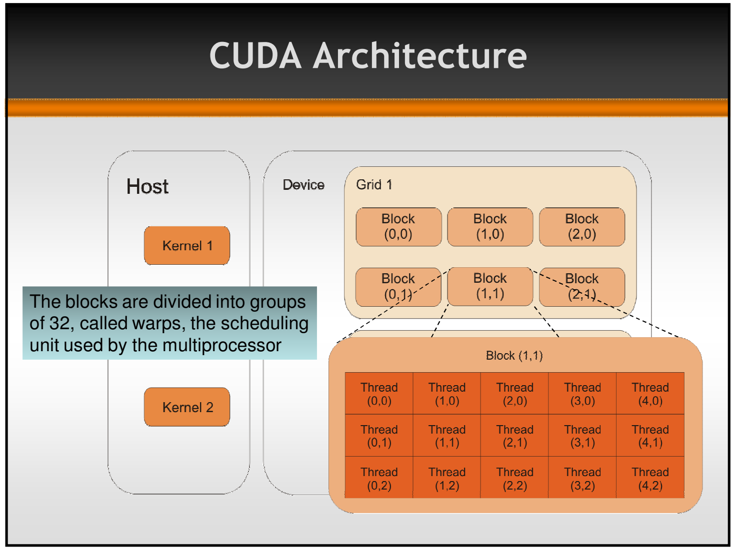


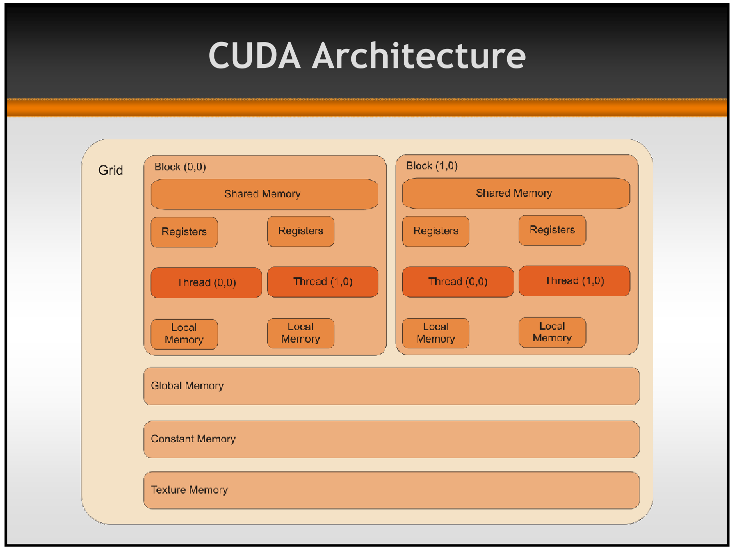
CUDA Architecture

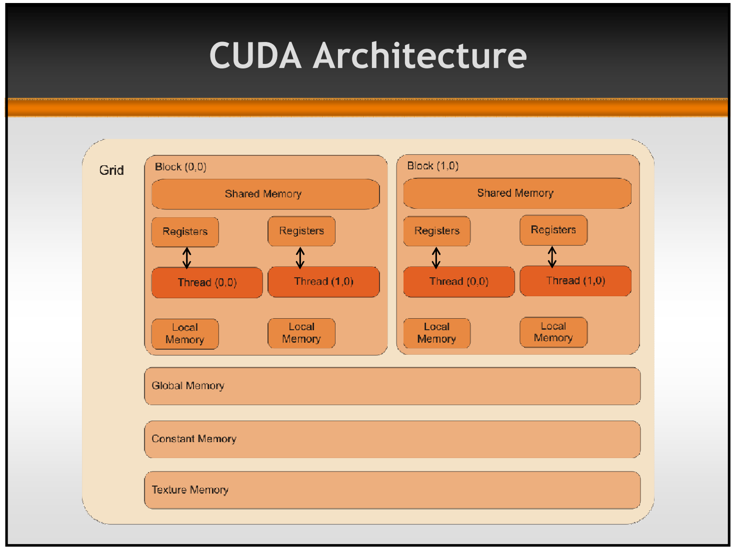


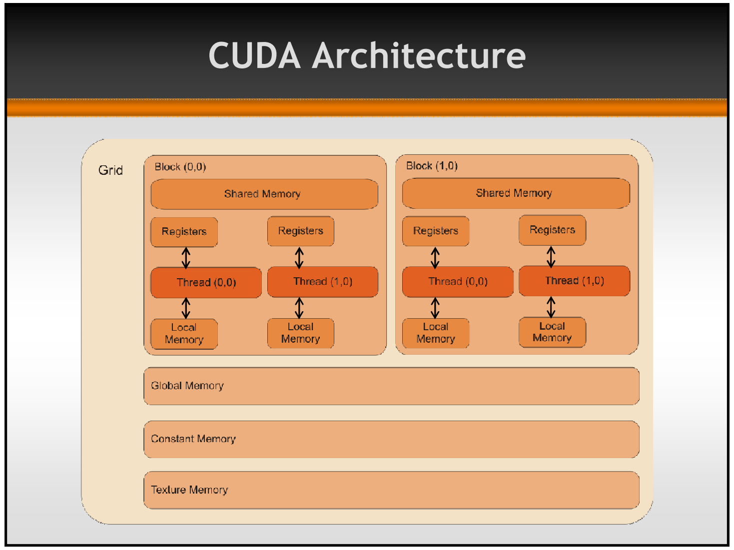
CUDA Architecture

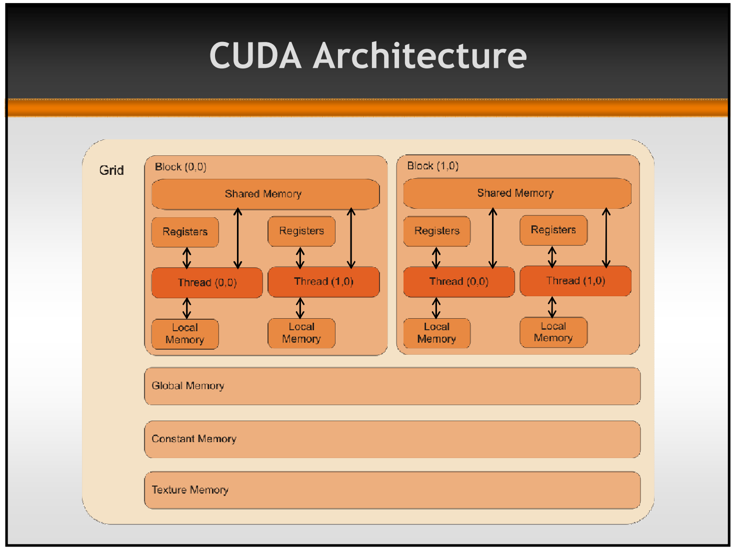


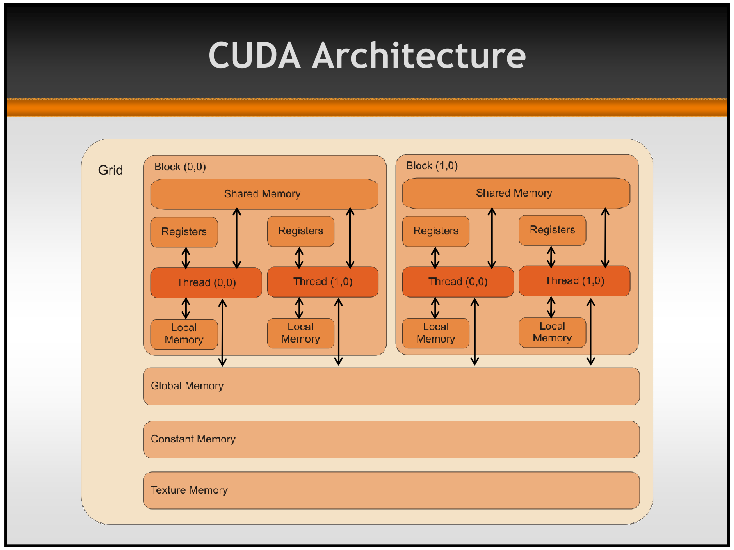

CUDA Architecture
Host
Kernel 1
Kernel 2
Device
Grid 1
Block (0,0)
Block (1,0)
Block (2,0)
Block (0,1)
Block (1,1)
Block (2,1)
The blocks are divided into groups of 32, called warps, the scheduling unit used by the multiprocessor
Block (1,1)
Thread (0,0)
Thread (1,0)
Thread (2,0)
Thread (3,0)
Thread (4,0)
Thread (0,1)
Thread (1,1)
Thread (2,1)
Thread (3,1)
Thread (4,1)
Thread (0,2)
Thread (1,2)
Thread (2,2)
Thread (3,2)
Thread (4,2)

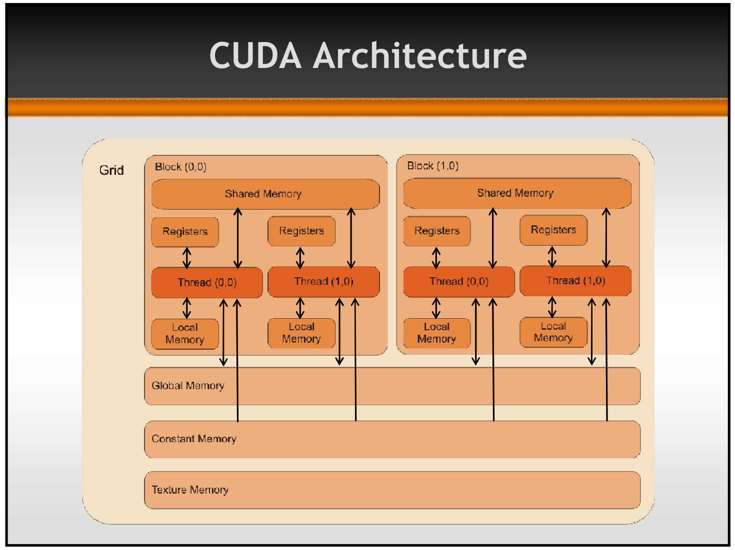

CUDA Architecture
Grid
Block (0,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Block (1,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory

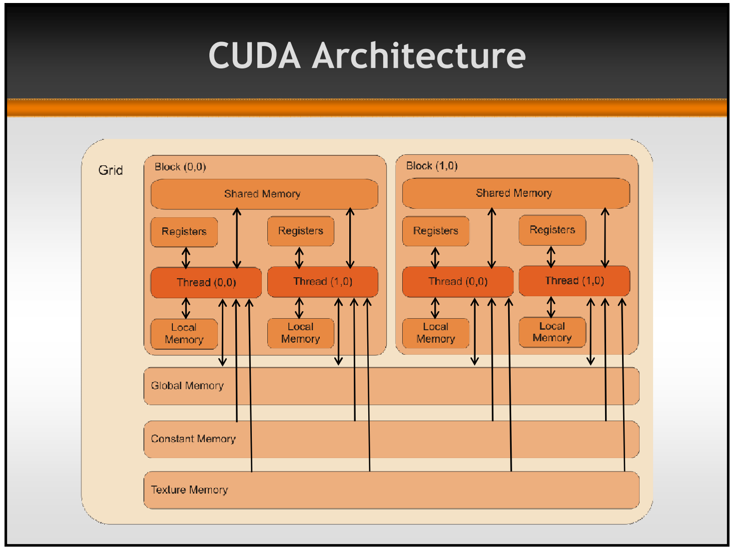

CUDA Architecture
Grid
Block (0,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Block (1,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory


CUDA Architecture
Grid
Block (0,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Block (1,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory


CUDA Architecture
Grid
Block (0,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Block (1,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory


CUDA Architecture
Grid
Block (0,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Block (1,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory


CUDA Architecture
Grid
Block (0,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Block (1,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory


CUDA Architecture
Grid
Block (0,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Block (1,0)
Shared Memory
Registers
Registers
Thread (0,0)
Thread (1,0)
Local Memory
Local Memory
Global Memory
Constant Memory
Texture Memory

CUDA Architecture

- "Crunching numbers" – On your head
  - Maximum number of threads per multiprocessor is 768, or 24 warps;
  - Threads must be organized in a maximum number of 8 blocks per-multiprocessor, and 512 threads per block;
  - Each multiprocessor contains 8192 32-bit registers, 16 KB of shared memory, 8 KB of cached constants and 8 KB of cached 1D textures.

CUDA Architecture

- G80 processor can be compared to a performance-optimized calculator; it is not as good as if there was a massive multi-core CPU
- Memory latency is a significant matter
  - The cost of memory access depends on its location. Local memory is several times slower than shared memory and cannot be cached
  - Local memory is a partition of the device memory, so it is important to use faster, on-chip, shared memory and registers

CUDA Architecture

- CUDA Occupancy Calculator
  - using just a few parameters, as threads per block, registers per thread and shared memory per block, the programmer can know how much he/she can improve CUDA applications
- CUDA profiler can also give kernel execution times in both GPU and CPU. The time spent with memory transfers is also monitored.

System Configuration

System Configuration

- CUDA is composed by three software:
  - CUDA SDK
  - CUDA Toolkit
  - CUDA graphics driver
- NVIDIA's CUDA Site
  - www.nvidia.com/object/cuda_home.html

System Configuration

- CUDA Developer SDK provides examples with source code, utilities, and white papers to help writing software with CUDA
- NVIDIA CUDA Toolkit contains the compiler, profiler, and additional libraries
  - CUBLAS (CUDA Basic Linear Algebra Subprograms)
  - CUFFT (CUDA Fast Fourier Transform)
  - PTX ISA (Parallel Thread Execution Instruction Set Architecture)

System Configuration

- Current versions of CUDA support only one version of the NVIDIA display driver
- Another issue on software versioning is the use of different versions of the SDK and Toolkit (i.e. SDK 2.0 with Toolkit 1.0), because of Application Programming Interface (API) and object code compatibility

System Configuration

- Compatible with: Windows XP, Mac OS X, Linux and Windows Vista
- Minimum hardware specs not defined by NVIDIA
  - 1 GB of system memory and at least 1 GB of free hard disk space would fit
  - To use a CUDA compatible based video card, it is necessary a vacant PCI-Express (1.0 or 2.0) slot and an additional specific power connector, depending on the model
- Natively compatible with Microsoft Visual Studio 2003 (7.0) and 2005 (8.0)

System Configuration

- CUDA is released free of charge for use in derivative works, whether academic, commercial, or personal
- Basically, it is prohibited to disassemble, decompile or reverse engineer the object code provided
- It is also determined that all NVIDIA copyright notices and trademarks should be acknowledged on derivative works, using the statement: “This software contains source code provided by NVIDIA Corporation”

CUDA Programming Approach

CUDA Programming Approach

- CUDA routines can be invoked from C/C++ code
  - Declaring the host functions with the `extern "C"` directive
  - Not using CUDA types in the function prototype
- Only CUDA host code is addressable by C/C++ files
  - Device code is addressable only by CUDA

CUDA Programming Model

- Programming model reflects the architecture
- Programming model concepts
  - Threads
  - Blocks
  - Grids
  - Kernel

Threads

- Smallest units
- Execute in parallel
- Grouped in blocks
- The minimum thread group that executes in parallel is called Warp and has WARP_SIZE = 32

Blocks

- Logically divided in 1, 2 or 3 dimensions
  - x, y, z
- Each dimension has a number of threads
- Threads belonging to the same block can synchronize among them
- Shared memory
  - 16KB
  - Faster than global memory
  - Great speed up

Grids

- Group of blocks
- Logically divided in 1, 2 or 3 dimensions
  - x, y, z
- Each dimension has a number of blocks
- The grid and block dimension and the amount of shared memory compose the kernel configuration

Kernel

- Code executed by each thread
- Needs a kernel configuration when invoked

Language Extensions

- Scope keywords
  - __device__ and __host__
    - Applied to variables and functions
  - __global__
    - Applied in kernel declaration
- Examples
  - `__device__ int number;`
  - `__host__ char c;`

Language Extensions

- Kernel invocation
  - Uses “<<<” and “>>>” to pass the configuration
  - Examples
    - kernel<<<128, 256>>>(params);
    - kernel<<<gridDim, blockDim>>>(params);
    - kernel<<< gridDim, blockDim, 1024>>>(params);

Language Extensions

- __shared__ keyword
  - Used to allocate the variable inside the shared memory space
  - Only threads belonging to the same block can access this variable
  - Each block has an instance of this variable
  - Example
    - `__shared__ char array[256];`

New Types

- Built-in vector types
  - All types except double have vector types
  - int2, uint3, float4, char4 etc.
  - Vector size is determined by the number in the type name
  - Elements are accessed as coordinates
    - x, y, z, w
  - dim3 type based on uint3
    - Values initialized with “1”
  - Example

```
float3 temp;
temp.x = 0.1f;
temp.y = 2.0f;
temp.z = 4.9f;
float4 f = make_float4(1.0f, 2.0f, 3.0f, 4.0f);
```

Templates

- Templates can be used in .cu files as in .cpp ones
- Can be applied to data and functions
- Allows compile-time pseudo-polimorphism
- Example

```
template<class T> T add3(T t1, T t2) {
   T result;
   result.x = t1.x + t2.x;
   result.y = t1.y + t2.y;
   result.z = t1.z + t2.z;
   return result;
}
```

Textures

- Cached memory access
- Any region of linear memory can be used as one-dimensional texture
- More than one dimension can be obtained using CUDA Arrays

Textures

- Texture references
  - Comunicates the host side with the device
  - Templates with type and dimension as parameters
  - Examples:
    - `texture<float, 1> texture1;`
    - `texture<int, 1> texture2;`

Using Textures

- cudaMalloc
- cudaMallocHost
- cudaMemcpy
- cudaMemset
- cudaBindTexture
- cudaUnbindTexture
- cudaFree
- cudaFreeHost

Additional Libraries

- CUFFT
  - Parallel Fast Fourier Transform
- CUBLAS
  - Numerical Algorithms

CUDA Programming Guidelines

Thread arrangement, Sequential and non-sequential memory access, Page-locked memory, Loop unrolling, Floating point conversion

Execution Configuration

- Qualifiers:
  - `__host__`, `__device__`, `__global__`
- Kernel declaration:

```
__global__ void kernelName(parameters) {
        ...
}
```

Execution Configuration

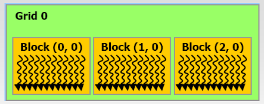
- Any call to a `__global__` function must specify the execution configuration for that call

<<< Dg, Db, Ns, S >>>

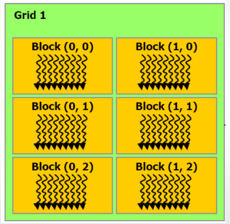
Dg: Grid dimensions
Db: Block dimensions
Ns: Number of bytes for shared memory
S: Stream associated to the kernel

Execution Configuration

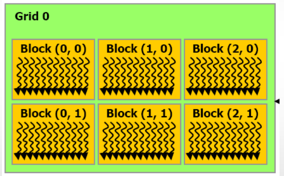
Grid dimensions
Block dimensions
} `dim3`



`dim3 Dg(3, 1, 1);`
or
`dim3 Dg(3);`

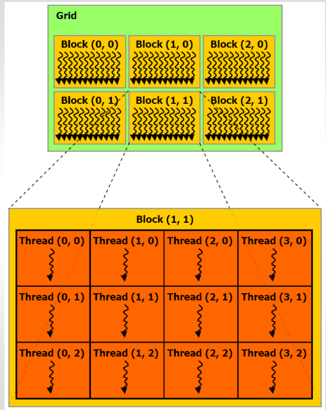


`dim3 Dg(2, 3, 1);`
or
`dim3 Dg(2, 3);`



`dim3 Dg(3, 2, 1);`
or
`dim3 Dg(3, 2);`

Execution Configuration



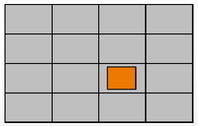

```
__global__ void Func(float* parameter);
```

```
dim3 Dg(3, 2);
dim3 Db(4, 3);

Func<<< Dg, Db >>>(parameter);
```

Execution Configuration

- Built-in Variables:
  `gridDim / blockDim / blockIdx / threadIdx`

```
unsigned int index = blockDim.x * blockIdx.x + threadIdx.x;
```



```
Func<<< 2, 8 >>>(parameter);
```

```
gridDim.x = 2;
blockDim.x = 8;
```

The third thread from second block will point to...

`blockIdx.x = 1; threadIdx.x = 2;`

Execution Configuration

256 threads

```
__global__ void bin1D(float* parameter) {

    unsigned int index = threadIdx.x;
    ...

}
```

16 threads

16 threads

```
__global__ void bin2D(float* parameter) {

    unsigned int index = threadIdx.y * 16 + threadIdx.x;
    ...

}
```

Thread Arrangement

```
__global__ void read_only_tex_1D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    __shared__ type shared[BLOCK_SIZE];
    shared[threadIdx.x] = tex1Dfetch(tex_##type, idx);
}
__global__ void read_only_tex_2D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    const unsigned int idy = threadIdx.y + __mul24(blockIdx.y, blockDim.y);
    const unsigned int index = threadIdx.x + __mul24(threadIdx.y,
        BLOCK_SIZE);

    __shared__ type shared[BLOCK_SIZE];
    shared[index] = tex2D(tex_2D_##type, idx, idy);
}
```

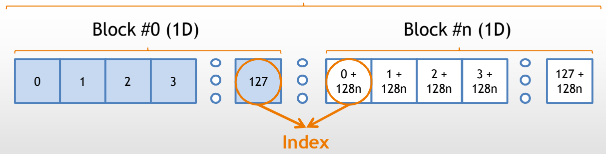
Read-only Texture Memory Kernel

Thread Arrangement

```
__global__ void read_only_tex_1D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    __shared__ type shared[BLOCK_SIZE];
    shared[threadIdx.x] = tex1Dfetch(tex_##type, idx);
}
```


```
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    const unsigned int idy = threadIdx.y + __mul24(blockIdx.y, blockDim.y);
    const unsigned int index = threadIdx.x + __mul24(threadIdx.y,
        BLOCK_SIZE);

    __shared__ type shared[BLOCK_SIZE];
    shared[index] = tex2D(tex_2D_##type, idx, idy);
}
```


Unidimensional Grid

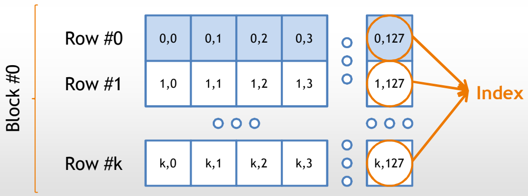


Texture Memory Access Pattern

Thread Arrangement


```
__global__ void read_only_tex_1D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    __shared__ type shared[BLOCK_SIZE];
```


```
__global__ void read_only_tex_2D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    const unsigned int idy = threadIdx.y + __mul24(blockIdx.y, blockDim.y);
    const unsigned int index = threadIdx.x + __mul24(threadIdx.y,
        BLOCK_SIZE);

    __shared__ type shared[BLOCK_SIZE];
    shared[index] = tex2D(tex_2D_##type, idx, idy);
}
```



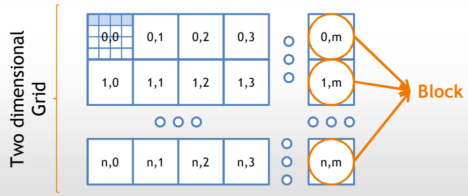
Texture Memory Access Pattern

Thread Arrangement


```
__global__ void read_only_tex_1D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    __shared__ type shared[BLOCK_SIZE];
```



```
__global__ void read_only_tex_2D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    const unsigned int idy = threadIdx.y + __mul24(blockIdx.y, blockDim.y);
    const unsigned int index = threadIdx.x + __mul24(threadIdx.y,
        BLOCK_SIZE);

    __shared__ type shared[BLOCK_SIZE];
    shared[index] = tex2D(tex_2D_##type, idx, idy);
}
```

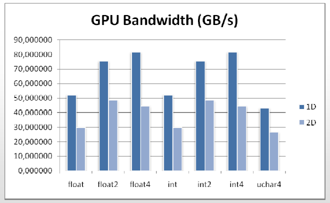
Texture Memory Access Pattern

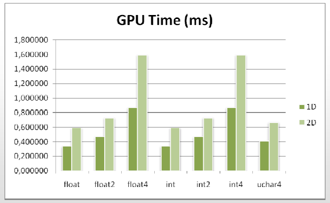
Thread Arrangement

```
__global__ void read_only_tex_1D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    __shared__ type shared[BLOCK_SIZE];
    shared[threadIdx.x] = tex1Dfetch(tex_##type, idx);
}
__global__ void read_only_tex_2D_##type() {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    const unsigned int idy = threadIdx.y + __mul24(blockIdx.y, blockDim.y);
    const unsigned int index = threadIdx.x + __mul24(threadIdx.y,
        BLOCK_SIZE);

    __shared__ type shared[BLOCK_SIZE];
    shared[index] = tex2D(tex_2D_##type, idx, idy);
}
```

Kernel Configuration

Number of elements: **4,456,448**
Threads per block: **128**
Unidimensional grid: **34,816 blocks**
Two dimensional grid: **16 x 2,176 blocks**





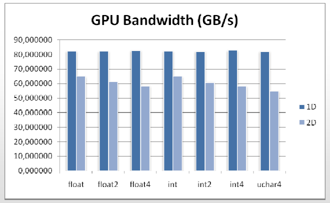
Thread Arrangement

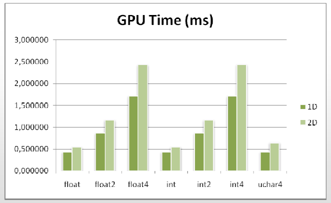
```
__global__ void copy_tex_1D_##type(type* g_odata) {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    g_odata[idx] = tex1Dfetch(tex_##type, idx);
}
__global__ void copy_tex_2D_##type(type* g_odata) {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    const unsigned int idy = threadIdx.y + __mul24(blockIdx.y, blockDim.y);
    g_odata[idx + __mul24(idy, __mul24(blockDim.x, gridDim.x))] =
        tex2D(tex_2D_##type, idx, idy);
}
```

Kernel Configuration

Number of elements: **4,456,448**
Threads per block: **128**
Unidimensional grid: **34,816 blocks**
Two dimensional grid: **16 x 2,176 blocks**

Copy from Texture Memory Kernel





Thread Arrangement

- One dimensional configurations (for both grids and blocks) proved to be the best thread arrangement
  - Less multiplications for index calculation

Recommended Guidelines

- One dimensional configurations (for both grids and blocks)

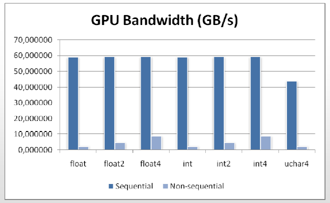
Sequential Memory Access

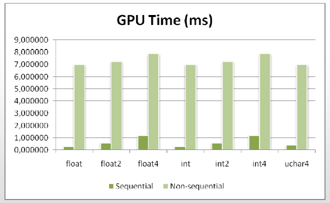
```
template <class T> __global__ void write_only(T* g_odata, T c) {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    g_odata[idx] = c;
}
template <class T> __global__ void write_only_t(T* g_odata, T c) {
    const unsigned int idxt = blockIdx.x + __mul24(threadIdx.x, gridDim.x);
    g_odata[idxt] = c;
}
```

Kernel Configuration

Number of elements: **4,456,448**
Threads per block: **128**
Unidimensional grid: **34,816 blocks**

Write-only Global Memory Kernel





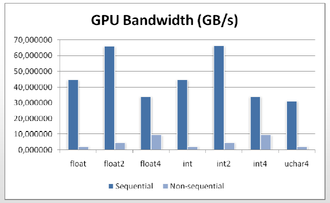
Sequential Memory Access

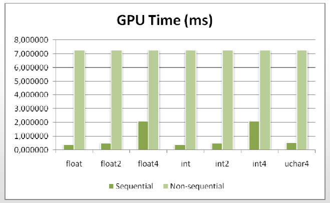
```
template <class T> __global__ void read_only_gmem(T* g_idata, T c) {
    const unsigned int idx = threadIdx.x + __mul24(blockIdx.x, blockDim.x);
    __shared__ T shared[BLOCK_SIZE];
    shared[threadIdx.x] = g_idata[idx];
}
template <class T> __global__ void read_only_gmem_t(T* g_idata, T c) {
    const unsigned int idxt = blockIdx.x + __mul24(threadIdx.x, gridDim.x);
    __shared__ T shared[BLOCK_SIZE];
    shared[threadIdx.x] = g_idata[idxt];
}
```

Kernel Configuration

Number of elements: **4,456,448**
Threads per block: **128**
Unidimensional grid: **34,816 blocks**

Read-only Global Memory Kernel





Sequential Memory Access

- Mandatory for higher performances
  - Use this guideline whenever possible (reads and writes)

Recommended Guidelines

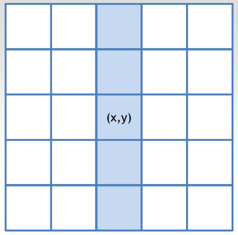
- One dimensional configurations (for both grids and blocks)
- Sequential reads and writes are mandatory for higher performances

Non-sequential Reading

```
__global__ void convolve_V5_##type(type* g_idata, type* g_odata, type c) {
    const unsigned int loadPos = (blockIdx.x << 7) + threadIdx.x;\
    const unsigned int y = (loadPos >> 7);\
    type sum = c;\
    if((y >= 2) && (y < (gridDim.x - 2))) {\
        sum = sum + (tex1Dfetch(tex_##type, loadPos - 2*128)\
            + tex1Dfetch(tex_##type, loadPos + 2*128))*0.0096200556f;\
        sum = sum + (tex1Dfetch(tex_##type, loadPos - 1*128)\
            + tex1Dfetch(tex_##type, loadPos + 1*128))*0.20542368f;\
        sum = sum + tex1Dfetch(tex_##type, loadPos)*0.56991249f;\
    }\
    g_odata[loadPos] = sum;\
}

template <class T>
__global__ void convolve_V5(T* g_idata, T* g_odata, T c) {
    const unsigned int loadPos = (blockIdx.x << 7) + threadIdx.x;
    const unsigned int y = (loadPos >> 7);
    T sum = c;
    if((y >= 2) && (y < (gridDim.x - 2))) {
        sum = sum + (g_idata[loadPos - 2*128]
            + g_idata[loadPos + 2*128])*0.0096200556f;
        sum = sum + (g_idata[loadPos - 1*128]
            + g_idata[loadPos + 1*128])*0.20542368f;
        sum = sum + g_idata[loadPos]*0.56991249f;
    }
    g_odata[loadPos] = sum;
}
```

Vertical Convolution Filter

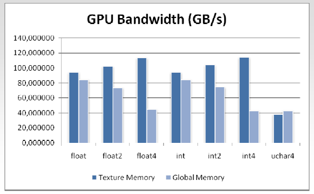
Non-sequential Reading

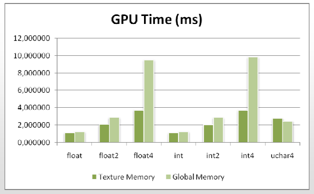


Memory Access Pattern

Kernel Configuration

Number of elements: **4,456,448**
Threads per block: **128**
Unidimensional grid: **34,816 blocks**





Non-sequential Reading

- Usage of textures could avoid the non-sequential reading bottleneck
  - Non-sequential positions must be close (in the 2D texture)

Recommended Guidelines

- One dimensional configurations (for both grids and blocks)
- Sequential reads and writes are mandatory for higher performances
- Usage of textures could avoid the non-sequential reading bottleneck

Shared Memory Usage

- Increase memory access speed
- Whenever more than one read from global memory is needed
- Threads are synchronized through the usage of `__syncthreads()` function
- Only 16KB per block

Recommended Guidelines

- One dimensional configurations (for both grids and blocks)
- Sequential reads and writes are mandatory for higher performances
- Usage of textures could avoid the non-sequential reading bottleneck
- Use shared memory whenever more than one read from global memory is needed

Page-locked Memory

- Device has direct access to host memory
  - No CPU polling
- Increased memory bandwidth
- Allocation through `cudaMallocHost` function
- Moderate usage should be done
  - The more page-locked memory is allocated, the fewer paged one is available, resulting in system performance degradation

Recommended Guidelines

- One dimensional configurations (for both grids and blocks)
- Sequential reads and writes are mandatory for higher performances
- Usage of textures could avoid the non-sequential reading bottleneck
- Use shared memory whenever more than one read from global memory is needed
- Page-locked memory increases host to device memory bandwidth transfer

Loop Unrolling

- Avoid branching tests
- More lines of code, but probable gain on performance
- It is highly dependent on the algorithm being implemented

```
// instead of doing this...
__global__ void sum_five(int* g_odata) {
    const unsigned int loadPos = threadIdx.x;
    int sum = 0;
    for (int i = 0; i < 5; i++)
        sum += tex1Dfecth(texture, loadPos + i);
    g_idata[loadPos] = sum;
}
```

```
// ... do this!
__global__ void sum_five_unrolled(int* g_odata) {
    const unsigned int loadPos = threadIdx.x;
    int sum = tex1Dfecth(texture, loadPos);
    sum += tex1Dfecth(texture, loadPos + 1);
    sum += tex1Dfecth(texture, loadPos + 2);
    sum += tex1Dfecth(texture, loadPos + 3);
    sum += tex1Dfecth(texture, loadPos + 4);
    g_idata[loadPos] = sum;
}
```

Recommended Guidelines

- One dimensional configurations (for both grids and blocks)
- Sequential reads and writes are mandatory for higher performances
- Usage of textures could avoid the non-sequential reading bottleneck
- Use shared memory whenever more than one read from global memory is needed
- Page-locked memory increases host to device memory bandwidth transfer
- Loop unrolling to decrease number of branches

Floating Point Conversion

- CUDA 1.0 compatible hardware does not support double precision
- Add the leading “f” to numbers
  - Instead of “1” or “1.0”, write “1.0f”
  - Do the same on host code!
- Avoid precision errors

Recommended Guidelines

- One dimensional configurations (for both grids and blocks)
- Sequential reads and writes are mandatory for higher performances
- Usage of textures could avoid the non-sequential reading bottleneck
- Use shared memory whenever more than one read from global memory is needed
- Page-locked memory increases host to device memory bandwidth transfer
- Loop unrolling to decrease number of branches
- Add the leading “f” to floating point numbers

Case Studies

Matrix transpose, Image convolution, Point Based Animation

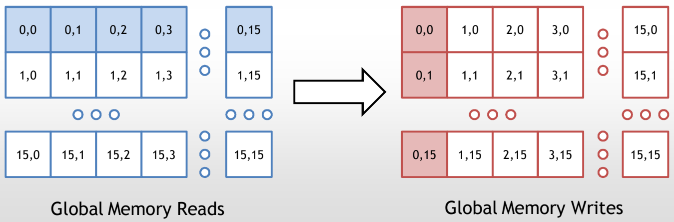
Transpose Matrix

- Highlights an interesting issue on implementation techniques
- Probably first developer thought
  - It could be done by simply computing, for each index, its transposed counterpart, and thus copying from one memory position to its destination
- Let's see what happens!

Transpose Matrix

```
__global__ void transpose_naive(float *odata, float* idata, int width,
int height) {
    unsigned int xIndex = __mul24(blockDim.x, blockIdx.x) + threadIdx.x;
    unsigned int yIndex = __mul24(blockDim.y, blockIdx.y) + threadIdx.y;

    if (xIndex < width && yIndex < height) {
        unsigned int index_in  = xIndex + width * yIndex;
        unsigned int index_out = yIndex + height * xIndex;
        odata[index_out] = idata[index_in];
    }
}
```

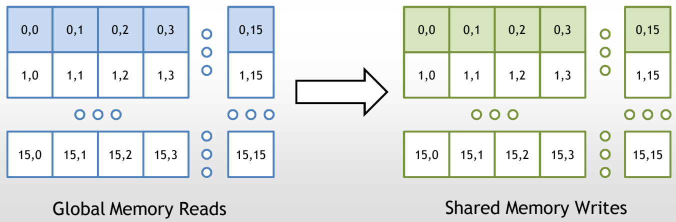
Naive Matrix Transposing



Transpose Matrix

```
if (xIndex < width && yIndex < height) {
        // load block into smem
        unsigned int index_in = __mul24(width, yIndex) + xIndex;
        unsigned int index_block = __mul24(threadIdx.y, BLOCK_DIM) + threadIdx.x;
        // load a block of data into shared memory
        block[index_block] = idata[index_in];
        index_transpose = __mul24(threadIdx.x, BLOCK_DIM) + threadIdx.y;
        index_out = __mul24(height, xBlock + threadIdx.y) + yBlock + threadIdx.x;
      }
      __syncthreads();
}
```

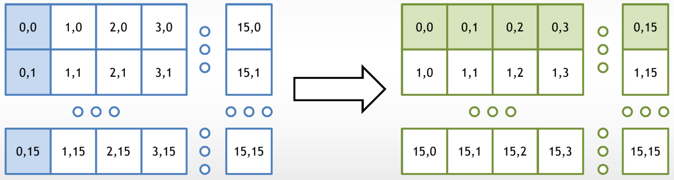
Smart Matrix Transposing (Copy to Shared Memory)



Transpose Matrix

```
if (xIndex < width && yIndex < height) {
    // write it out (transposed) into the new location
    odata[index_out] = block[index_transpose];
}
```

Smart Matrix Transposing (Write to Global Memory)



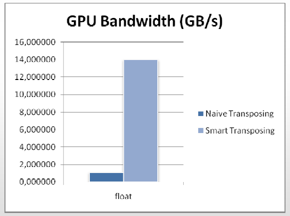
Shared Memory Reads

Global Memory Writes

Transpose Matrix

Kernel Configuration

Number of elements: **8,192 x 8,192 = 67,108,864**
Threads per block: **16 x 16 threads**
Two dimensional grid: **512 x 512 blocks**



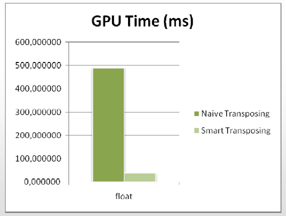


Image Convolution

Image Convolution

- Multi-purpose algorithm used for edge detection, smoothing, noise reduction, etc.
- Weights to be applied to pixels within a window surrounding the output pixels

Image Convolution

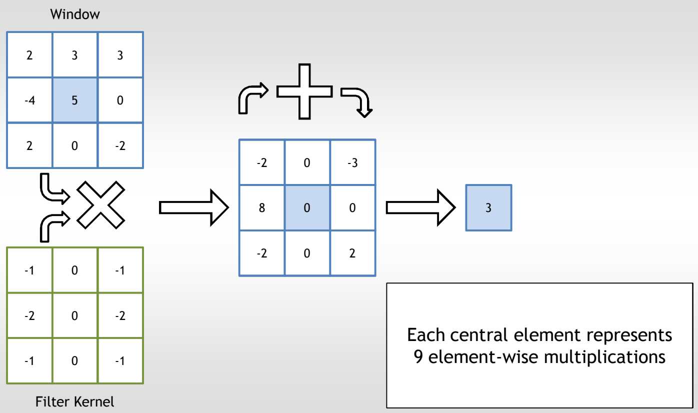


Image Convolution

- Some filters, called separable filters, can be split in two ones
- Each filter is applied separately
- Instead of doing `n*m` multiplications, only `n+m` are necessary

Image Convolution

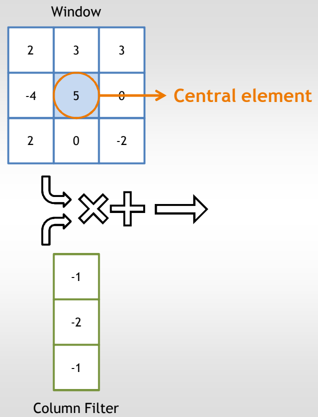
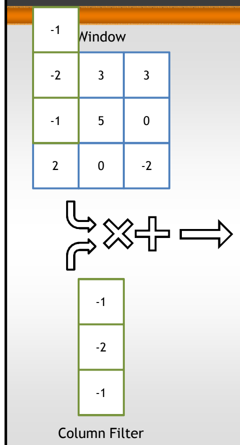


Image Convolution



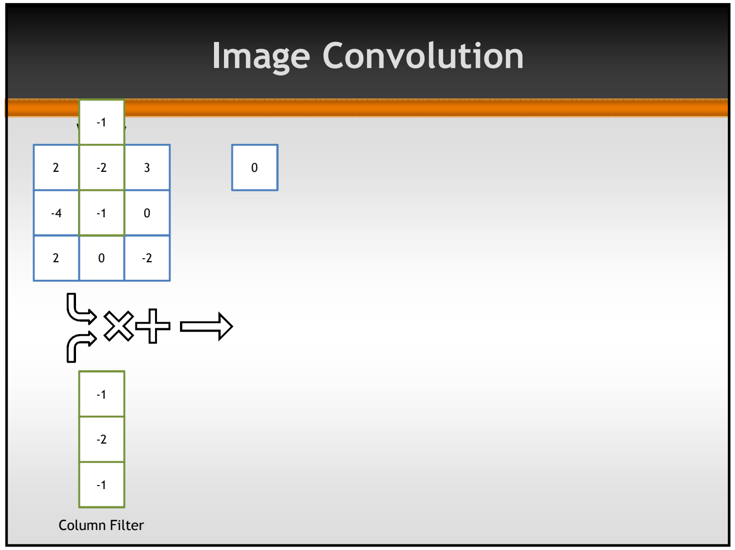


Image Convolution
-1
2 -2 3
-4 -1 0
2 0 -2
0
-1
-2
-1
Column Filter

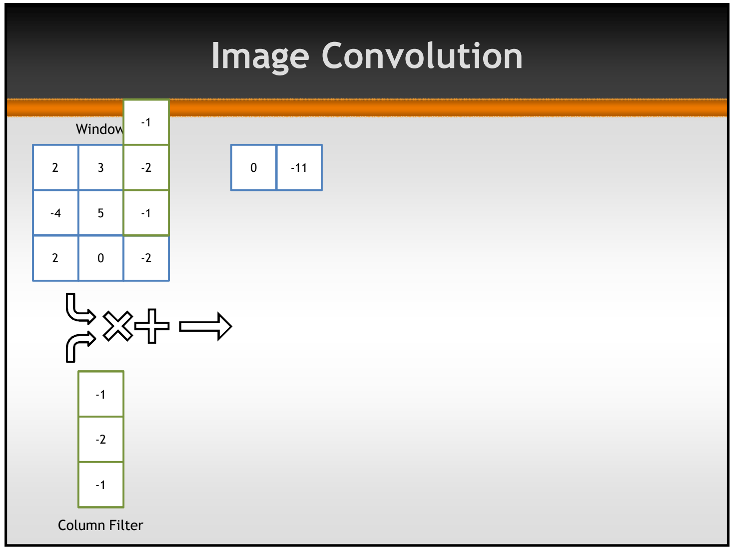


Image Convolution
Window
-1
2 3 -2
-4 5 -1
2 0 -2
0 -11
-1
-2
-1
Column Filter

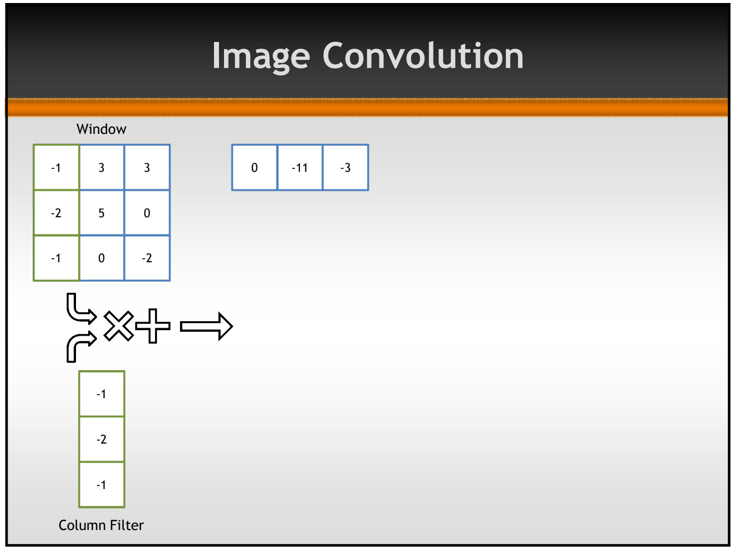


Image Convolution
Window
-1 3 3
-2 5 0
-1 0 -2
0 -11 -3
-1
-2
-1
Column Filter

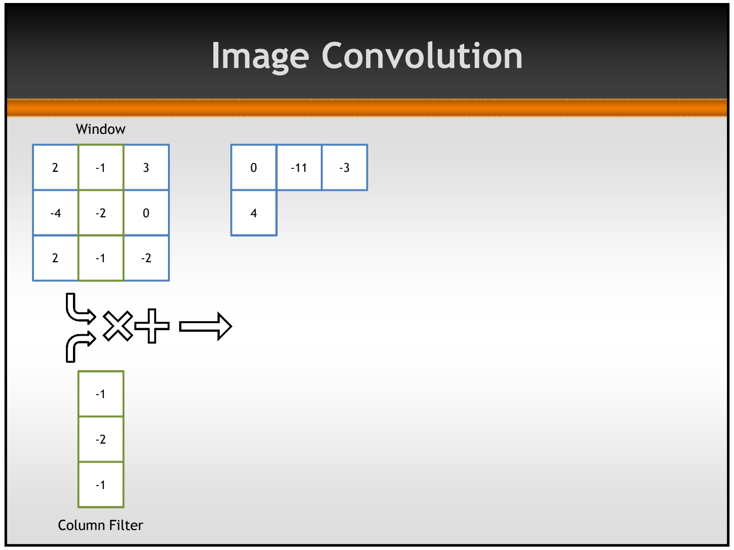


Image Convolution
Window
2 -1 3
-4 -2 0
2 -1 -2
0 -11 -3
4
-1
-2
-1
Column Filter

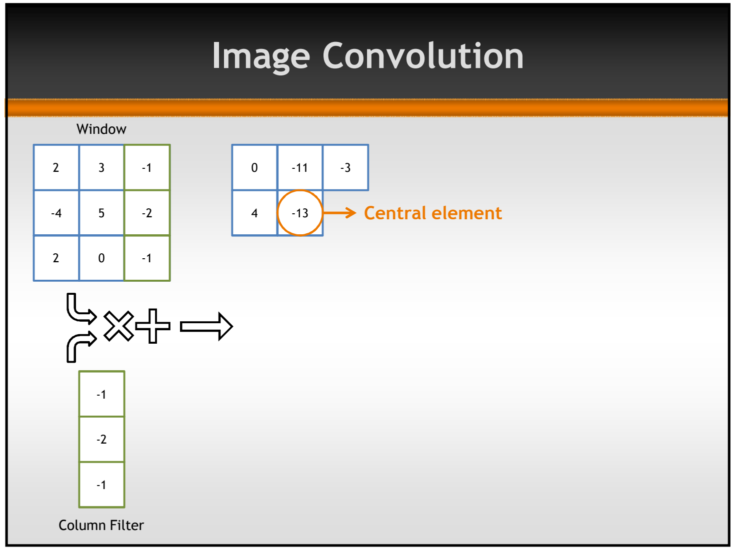


Image Convolution
Window
2 3 -1
-4 5 -2
2 0 -1
0 -11 -3
4 -13
Central element
-1
-2
-1
Column Filter

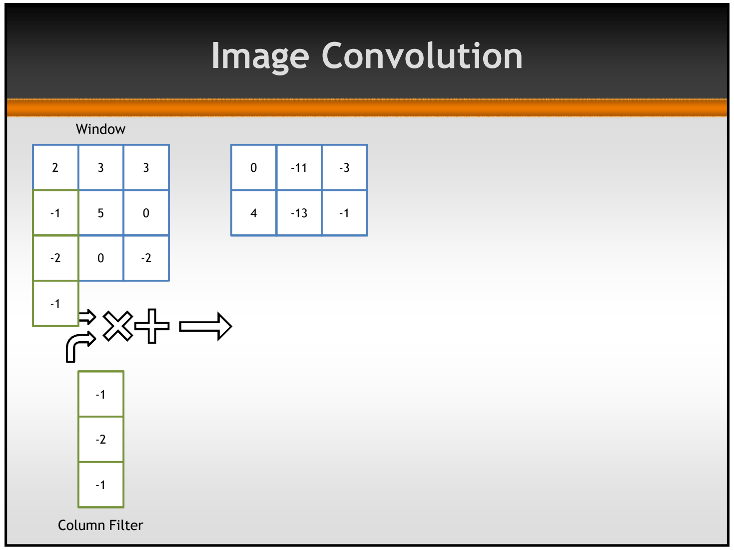


Image Convolution
Window
2 3 3
-1 5 0
-2 0 -2
-1
0 -11 -3
4 -13 -1
-1
-2
-1
Column Filter

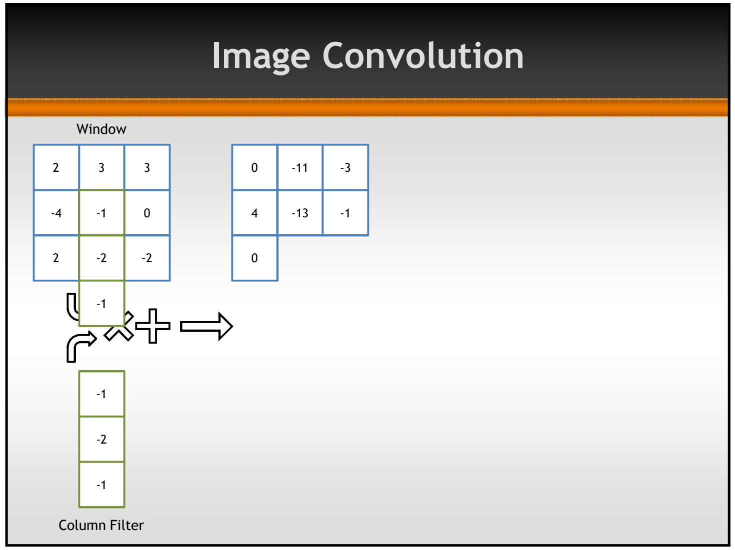


Image Convolution
Window
2 3 3
-4 -1 0
2 -2 -2
-1
0 -11 -3
4 -13 -1
0
-1
-2
-1
Column Filter

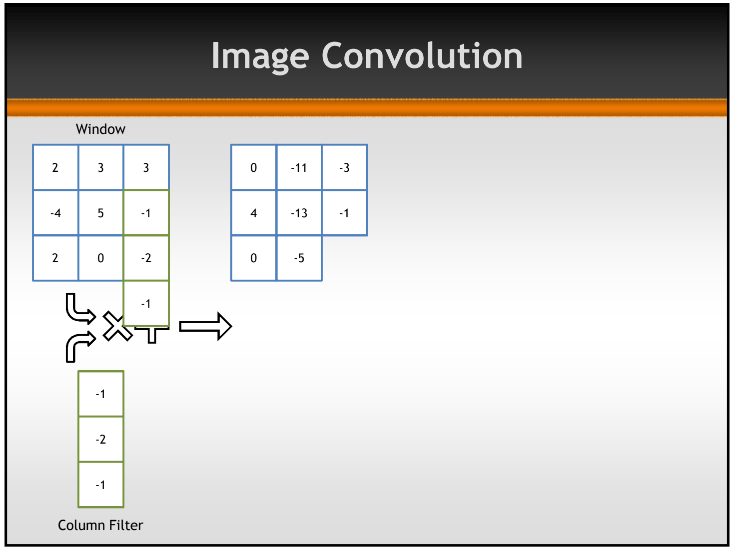


Image Convolution
Window
2 3 3
-4 5 -1
2 0 -2
-1
0 -11 -3
4 -13 -1
0 -5
-1
-2
-1
Column Filter

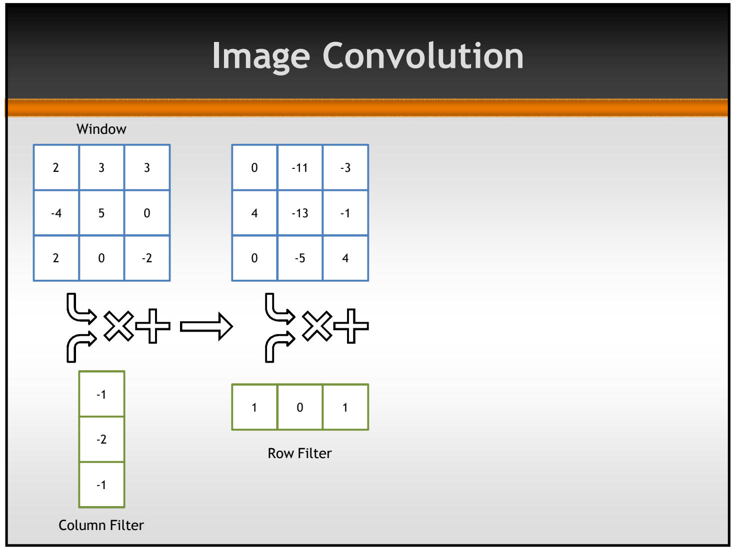


Image Convolution
Window
2 3 3
-4 5 0
2 0 -2
0 -11 -3
4 -13 -1
0 -5 4
1 0 1
Row Filter
-1
-2
-1
Column Filter

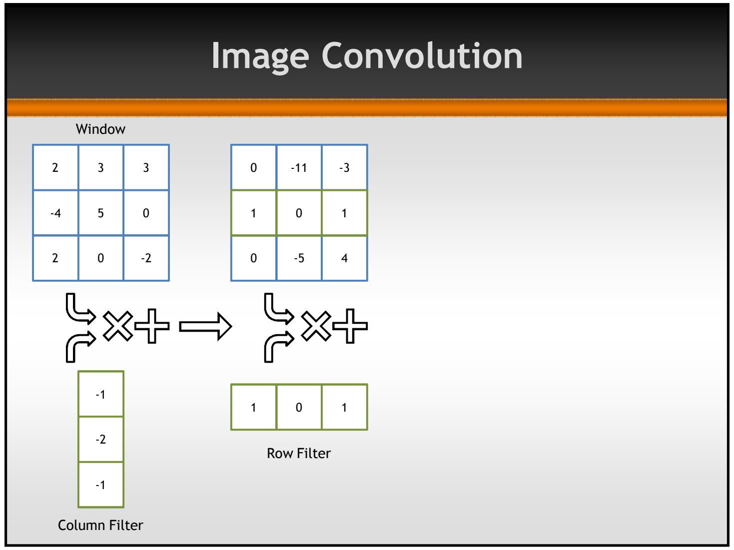


Image Convolution
Window
2 3 3
-4 5 0
2 0 -2
0 -11 -3
1 0 1
0 -5 4
1 0 1
Row Filter
-1
-2
-1
Column Filter

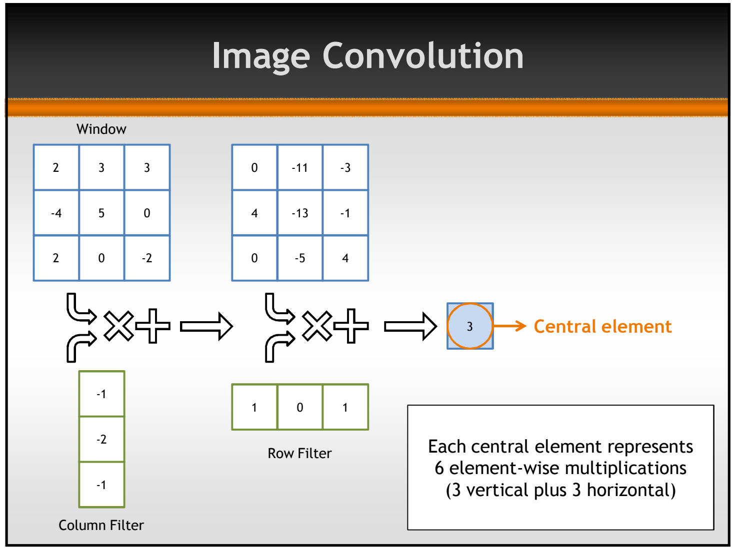


Image Convolution
Window
2 3 3
-4 5 0
2 0 -2
0 -11 -3
4 -13 -1
0 -5 4
3
Central element
1 0 1
Row Filter
-1
-2
-1
Column Filter
Each central element represents 6 element-wise multiplications (3 vertical plus 3 horizontal)

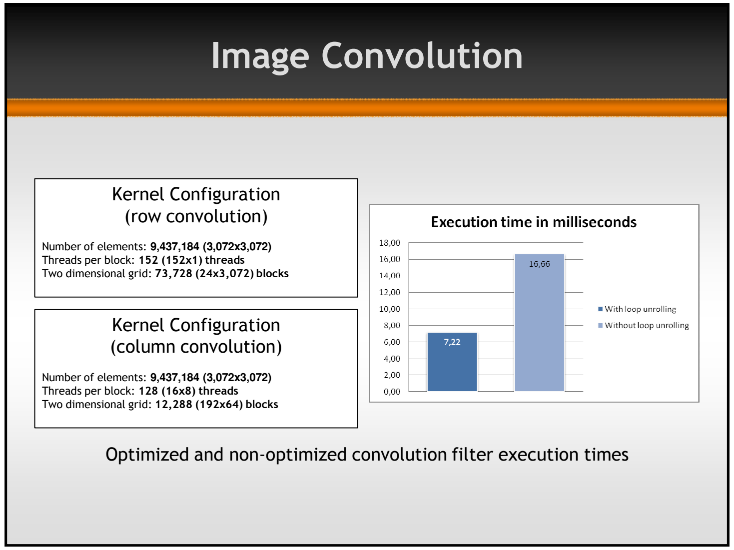
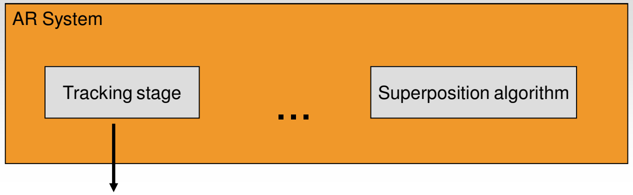


Image Convolution
Kernel Configuration (row convolution)
Number of elements: 9,437,184 (3,072x3,072)
Threads per block: 152 (152x1) threads
Two dimensional grid: 73,728 (24x3,072) blocks
Kernel Configuration (column convolution)
Number of elements: 9,437,184 (3,072x3,072)
Threads per block: 128 (16x8) threads
Two dimensional grid: 12,288 (192x64) blocks
Execution time in milliseconds
18,00
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
7,22
16,66
With loop unrolling
Without loop unrolling
Optimized and non-optimized convolution filter execution times

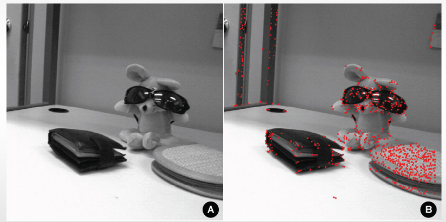
KLT Tracker



- Edge detection
- Template matching
- Scale invariant features (SIFT)
- Optical flow

KLT Tracker

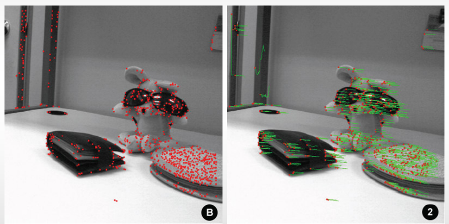
- Kanade Lucas Tomasi Tracker
  - Good Features to Track (GFTT)
  - Tracking stage



Input Image GFTT Result

KLT Tracker

- Kanade Lucas Tomasi Tracker
  - Good Features to Track (GFTT)
  - Tracking stage

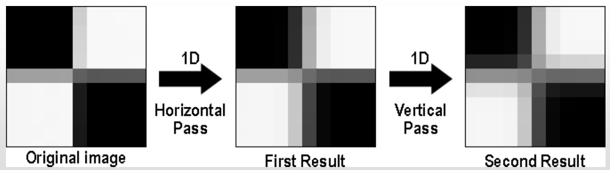


GFTT Result Tracking Result

KLT Tracker

- Implementation :: GFTT

```
cudaBindTexture(0, convolve_tex, d_img, SIZE_BYTES);
convolveH5<<< 1 << 13, 144 >>>(d_tmp_img1);
cudaThreadSynchronize();
cudaUnbindTexture(convolve_tex);

cudaBindTexture(0, convolve_tex, d_tmp_img1, SIZE_BYTES);
convolveV5<<< 1 << 13, 128 >>>(d_tmp_img2);
cudaThreadSynchronize();
cudaUnbindTexture(convolve_tex);
```



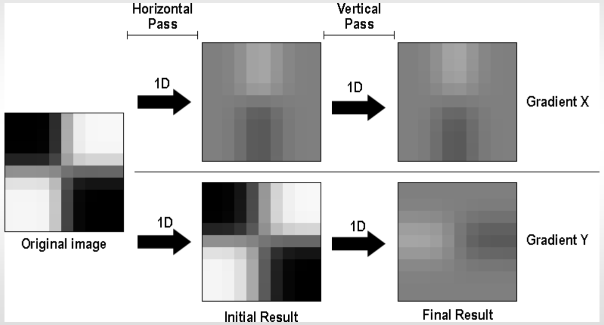
KLT Tracker

- Implementation :: GFTT

```
__global__ void convolve7both1s(float* d_gradx, float* d_grady) {
  __shared__ float tmp[3 + 128 + 3 + 10];
  const unsigned int loadPos = (blockIdx.x << POT_THREADS) + threadIdx.x;
  const int x = (loadPos & (WIDTH - 1)) - 3;

  tmp[threadIdx.x] = tex1Dfetch(convolve_tex, loadPos - 3);
  __syncthreads();

  if(threadIdx.x < 128) {
    float sum1 = 0.0f;
    float sum2 = 0.0f;
    if((x >= 0) && (x < (WIDTH - 3 - 3))) {
      sum1 += (tmp[threadIdx.x+6]-tmp[threadIdx.x])*0.013353735f;
      sum1 += (tmp[threadIdx.x+5]-tmp[threadIdx.x+1])*0.10845453f;
      sum1 += (tmp[threadIdx.x+4]-tmp[threadIdx.x+2])*0.24302973f;

      sum2 += (tmp[threadIdx.x] + tmp[threadIdx.x+6])*0.0044330480f;
      sum2 += (tmp[threadIdx.x+1] + tmp[threadIdx.x+5])*0.054005578f;
      sum2 += (tmp[threadIdx.x+2] + tmp[threadIdx.x+4])*0.24203622f;
      sum2 += tmp[threadIdx.x+3]*0.39905027f;
    }
    d_gradx[loadPos] = sum1;
    d_grady[loadPos] = sum2;
  }
}
```

KLT Tracker

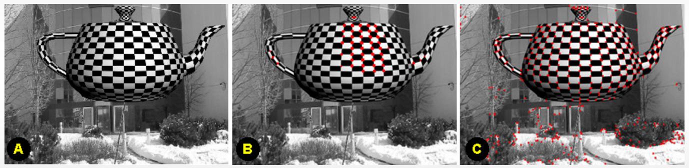
- Implementation :: GFTT



KLT Tracker

- Implementation :: GFTT

$$mineigenvalue = \frac{gxx + gyy - \sqrt{(gxx - gyy)^2 + 4gxy^2}}{2}$$



KLT Tracker

- Implementation :: GFTT

```
#define IFMAX2TEMP if(temp.x >= max2.x) max2 = temp;
#define __unroll_loop_enforce(i) \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10)]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 1]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 2]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 3]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 4]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 5]; IFMAX2TEMP \
  temp = sdata[threadIdx.x + (i)*(3+128+3 + 10) + 6]; IFMAX2TEMP

...

float2 temp;
float2 max2 = make_float2(0.0f, 0.0f);
__unroll_loop_enforce(0)
__unroll_loop_enforce(1)
__unroll_loop_enforce(2)
__unroll_loop_enforce(3)
__unroll_loop_enforce(4)
__unroll_loop_enforce(5)
__unroll_loop_enforce(6)

if(p.x != max2.x) {
  features[pos].x = 0.0f;
} else {
  if(max2.y > p.y) {
    features[pos].x = 0.0f;
  } else {
    features[pos] = p;
  }
}
```

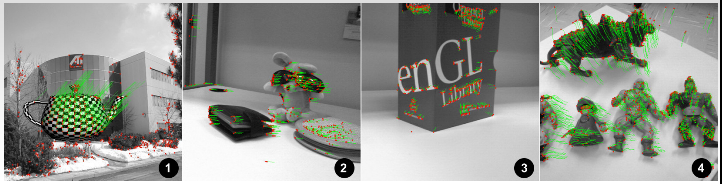
KLT Tracker

- Implementation :: Tracking stage
  - Pyramid calculation
  - Search correspondence inside window

Pyramid calculation step by step

Step	Description
Convolve image	5x5 gaussian filter
Calculate gradients	partial derivative 7x7 filter
Convolve image (subsampling 1/4)	21x21 gaussian filter
Calculate gradients (subsampling 1/4)	partial derivative 7x7 filter

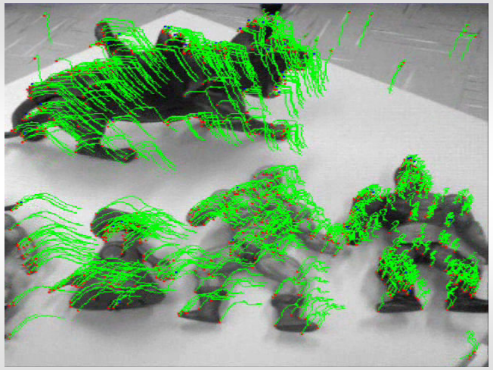
KLT Tracker



GPU processing times (in milliseconds) for all scenarios

	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Good Features To Track	47.54431	47.72869	47.67058	47.69209
Track Features (first time)	22.45760	27.47164	26.67517	21.68404
Track Features (remaining times)	21.91172	23.60970	22.36178	25.39792
Reselect Features	19.50471	14.36439	15.22232	15.00386

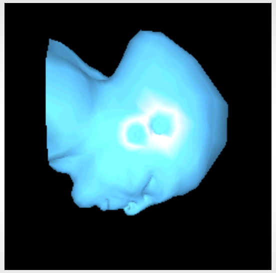
KLT Tracker



Resolution: 1024x1024
Number of features: 1000
Frames per second: ~50

Point Based Animation (PBA)

- Physics Models
  - Accurate Simulation
    - Offline
    - Originally Tetrahedrons
  - Mass-Spring Systems
    - Lighter Models



PBA

- Finite Element Based
- Mesh-Free Approach
- Continuum Mechanics Concepts
- Potential Capabilities
  - Elastic, Plastic and Melting Objects
  - Topological Changes
  - Rendering Flexibility

Why PBA?

- Continuum Mechanics
  - Many Particles
  - Same Properties
  - Suitable Applications
    - Fluid Mechanics
    - Civil Engineering

Why PBA?

- Particles
  - Discrete Set of Points
  - Meshless
  - Without Connectivity
    - Processed Separately
    - Little Information Needed
  - Stress and Strain

Why PBA?

- Physics Concepts
  - Accurate Modeling
    - PhysX
  - Highly Parallelizable
  - Real Time

How PBA Works

- Physics Elements (Phyxels)
  - Simulation Quantities
    - Position, Displacement, Velocity
  - Reference Shape
  - Support Radius
    - Mass, Volume and Density
- Initialization
  - Momentum Matrix

How PBA Works

- Add External Forces
- Calculate Strain and Stress
  - For all Neighboring Phyxels
    - Spatial Hash
  - Young's Modulus
  - Poisson's Ratio
- Update forces
  - Action and Reaction

Strain and Stress

- Strain ($\Delta l/l$)
  - Displacement Field
  - Not scalar like 1D case
  - $\mathbf{u}(u, v, w)^T$
- Stress (f/A)
  - Linearly related
  - $\sigma = \mathbf{E}\cdot\varepsilon$

$$\epsilon = \begin{bmatrix} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ \epsilon_{xy} & \epsilon_{yy} & \epsilon_{yz} \\ \epsilon_{xz} & \epsilon_{yz} & \epsilon_{zz} \end{bmatrix}$$

$$\sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{xy} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{xz} & \sigma_{yz} & \sigma_{zz} \end{bmatrix}$$

Hooke's Law

$$\begin{bmatrix} \sigma_{xx} \\ \sigma_{yy} \\ \sigma_{zz} \\ \sigma_{xy} \\ \sigma_{yz} \\ \sigma_{zx} \end{bmatrix} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & 1-2\nu & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-2\nu & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-2\nu \end{bmatrix} \begin{bmatrix} \epsilon_{xx} \\ \epsilon_{yy} \\ \epsilon_{zz} \\ \epsilon_{xy} \\ \epsilon_{yz} \\ \epsilon_{zx} \end{bmatrix}$$

Constants

- Young's Modulus (E)
  - Proportionality Constant
  - Material Dependent
    - Steel: 10^{11} N/m²
    - Rubber: ~10^{7} N/m²
- Poisson's Ratio (ν)
  - [0 .. ½)
  - Volume Conservation

Integration

- Explicit Integration
  - Euler
  - Verlet
  - Runge-Kutta 4th Order
- Implicit Integration
  - In Progress…

Rendering

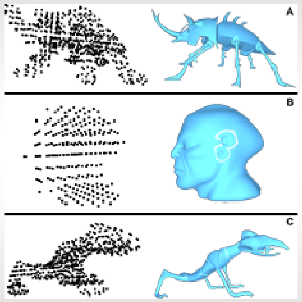
- Passive Surfel Advection
  - Interpolation of Surfel's Displacement Vectors
  - Nearby Phyxels
  - Mesh Vertices as Surfels
  - $\mathbf{u}_{sfl} = \frac{1}{\sum_i \omega(r_i, h)} \sum_i \omega(r_i, h)(\mathbf{u}_i + \nabla \mathbf{u}_i^T(\mathbf{x}_{sfl} - \mathbf{x}_i))$
- Point Based Approach
  - In Progress…

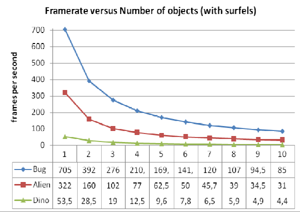
CUDA Implementation

- 5 kernels:
  - *precalcImutableValues*
  - *calcStrainStressA*
  - *calcStrainStressB*
  - *Integrate*
  - *updateSurfels*

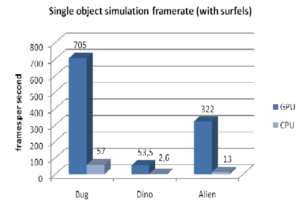
```
struct CuBody {
        float *d_positions_masses;
        float *d_displacements_volume;
        float *d_forces;
        float *d_momentMatrices;
        int numPhyxels;
        int maxNeighbors;
        unsigned short *d_neighbors;
        float *d_omega;
        float *d_gradui;
        float *d_lastDisplacements;
};
```

PBA Results





	1	2	3	4	5	6	7	8	9	10
Bug	705	392	276	210,	169,	141,	120	107	94,5	85
Alien	322	160	102	77	62,5	50	45,7	39	34,5	31
Dino	53,5	28,5	19	12,5	9,6	7,8	6,5	5,9	4,9	4,4



Final Considerations

Final Considerations

- GPGPU technology applies to MAR related problems
  - important contributions related to interest point based techniques and tracking of corners and edges, implemented using this technology
- Massive data processing applications have for a long time demanded expensive dedicated hardware to run. This new approach should bring image processing of HD videos to the desktop
- Using this approach, we can unify the CPU and GPU programming, and maintain time costly algorithms running concurrently with a sophisticated HD MAR pipeline

? !

Additional material available at:
http://www.gprt.ufpe.br/~grvm/pba

CUDA as a Supporting Technology for Next-Generation AR Applications

Thiago Farias, João Marcelo Teixeira, Pedro Leite,
Gabriel Almeida, Veronica Teichrieb, Judith Kelner
{tsmcf, jmnxt, pjsl, gfa, vt, jk}@cin.ufpe.br

Virtual Reality and Multimedia Research Group
Federal University of Pernambuco, Computer Science Center

9/19/2008