Universidade Federal de Santa Maria
Departamento de Eletrônica e Computação
Prof. Cesar Tadeu Pozzer
Disciplina: Computação Gráfica Avançada
pozzer@inf.ufsm.br
11/12/2008

Elaboração: Cesar Pozzer e Victor Chitolina Schetinger

CUDA - Conceitos Básicos

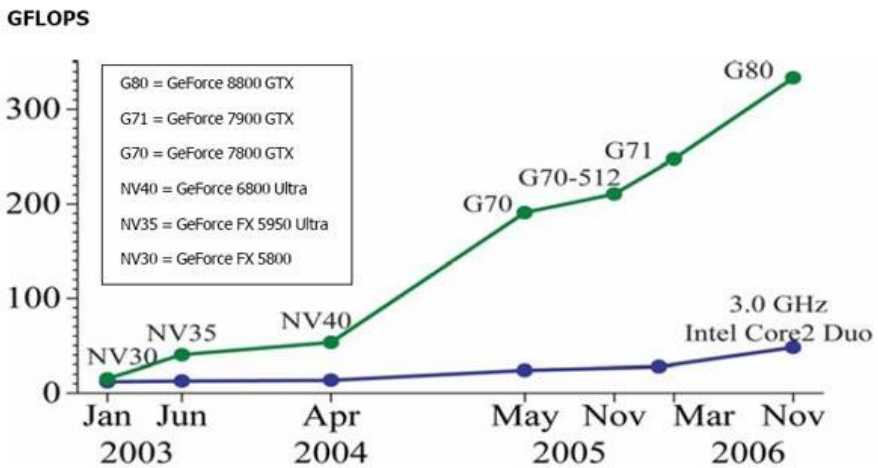
1. Computação Genérica em GPU

Diversas etapas do pipeline gráfico podem processar dados em paralelo (SIMD – *Single Instruction Multiple Data*), como por exemplo transformações de vértices, ou geração da cor de cada pixel. Por isso, o hardware gráfico é altamente paralelizável, o que pode ser observado pelo número de vertex/pixels/stream processors presentes em cada GPU.

Quando um programa de shader apresenta desvios condicionais, o processor associado executa tanto o If como o else, para evitar uma quebra de fluxo. Após a execução, decide qual resultado utilizar. Deve-se lembrar que todos os stream processors executam o mesmo código simultaneamente. A plataforma Tesla apresenta uma arquitetura baseada em SIMT (*Single Instruction Multiple Thread*), que permite que threads sejam executadas concorrentemente.

Devido a este paralelismo, o desempenho, medido em FLOPs, de GPUs é muito superior que o das CPUs atuais, como pode ser visto na seguinte figura. Atualmente as placas de vídeo mais modernas já tem desempenho superior a 1 teraFLOPS (no ano de 2008). Por exemplo, um Pentium IV Dual Core possui "ínfimos" 28 gigaFLOPS.

A seguinte tabela mostra um comparativo da evolução de desempenho entre CPUs e GPUs.



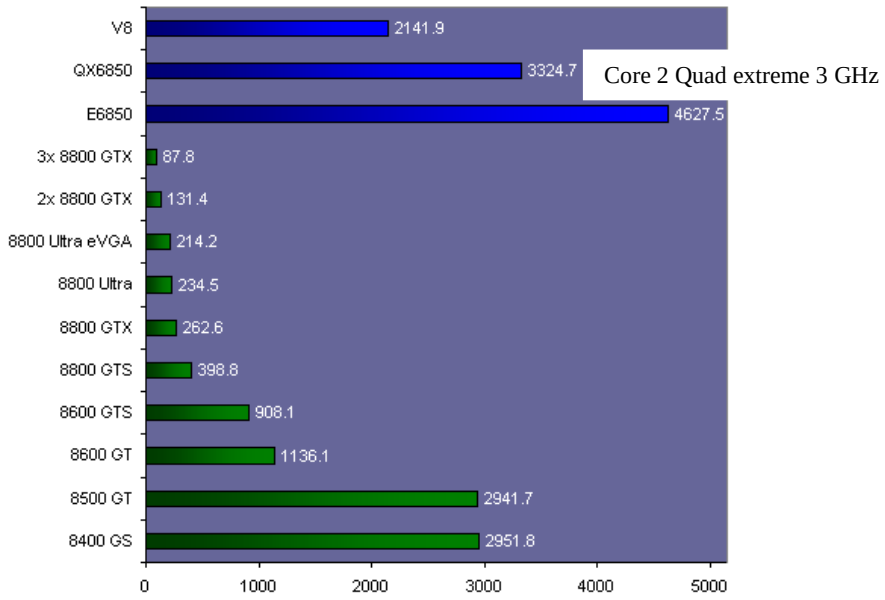
http://www.hwupgrade.it/articoli/skvideo/1750/nvidia-tesla-gpu-g80-per-elaborazioni-gpgpu_5.html

Desta forma começou-se a realizar pesquisas para utilizar as GPUs como co-processadores numéricos da CPU. Isso foi possível com o advento de GPUs programáveis, e linguagens para estas GPUs como ocorre por exemplo com a linguagem GLSL. Como esta linguagem foi desenvolvida para processamento gráfico específico, o seu uso para processamento genérico é mais complicado e exige habilidades do programador. Por exemplo, pode passar dados via textura (codificados em RGG, por exemplo), processa-se a informação, recupera-se a informação do frame buffer e decodifica-se a informação para o problema em questão.

2. CUDA

Observando esta tendência de mercado, a Nvidia criou o CUDA ("*Compute Unified Device Architecture*"), uma tecnologia GPGPU (*General Purspose* – GPU) que permite o programador usar a linguagem C para codificar algoritmos para execução na GPU. Atualmente, somente a série 8000 da Nvidia ou superiores suportam esta tecnologia. Está disponível um SDK com exemplos e código fonte para diversos problemas que podem ser resolvidos de forma vetorial em GPU, incluído, por exemplo, multiplicação de matrizes, cálculo de matriz transposta, convolução de imagens, FFT, dentre muitos outros. Para maiores informações, consulte o site http://www.nvidia.com/object/cuda_learn.html.

No seguinte gráfico ilustra-se a diferença de desempenho de três máquinas top da Intel para um problema específico: um Core 2 Duo 3GHz, um Core 2 Extreme QX6850 3GHz (4 núcleos), e um Dual Xeon X5365s (quadcore, 3 GHz) (8 núcleos), com GPUs da série 8000 da nVidia. (http://www.behardware.com/articles/678-7/nvidia-cuda-practical-uses.html). Observa-se que mesmo uma geForce 8400 supera um Core Quad Extreme e que o desempenho aumenta de forma quase linear quando utiliza-se mais de uma placa de vídeo.



Na seguinte figura são ilustradas placas top de linha em 2008 com suporte a CUDA. A versão a) não tem suporte a APIs gráficas, mas tem um desempenho muito superior em CUDA que o modelo b), que tem todo o pipeline gráfico e também suporte a CUDA.

a) Tesla C1060 b) GeForce GTX 280

3. A Programação em CUDA

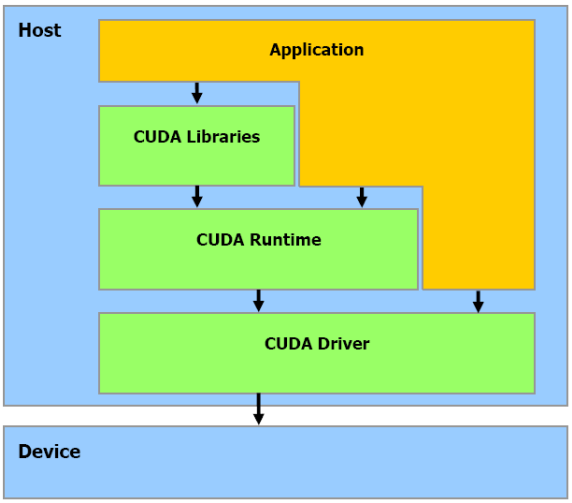
Um dispositivo GPU possui uma arquitetura muito diferente de uma CPU, e isso afeta diretamente a programação do mesmo. Primariamente, GPUs são extremamente paralelos, possuem mais de um núcleo e uma imensa quantidade de ULAs menos sofisticados que uma CPU, diversas caches e uma memória dedicada que é compartilhada por todos os núcleos.

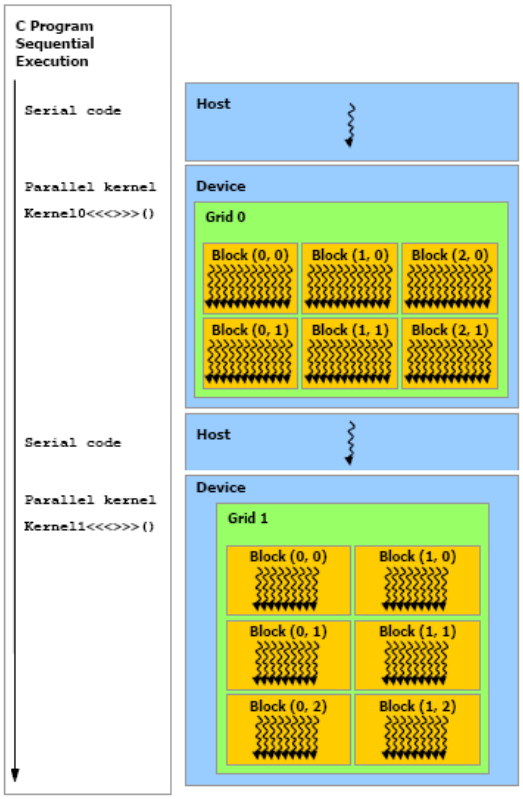
A GPU também se enquadra em um modelo SIMD de processamento, o que significa que o mesmo conjunto de instruções será executado paralelamente em todos processadores. Isso implica na necessidade de as aplicações desenvolvidas serem obviamente paralelizáveis, ou seja, qualquer solução que não se enquadra nesta especificação simplesmente não é interessante de ser desenvolvida em CUDA, ou por excesso de complexidade ou por baixo ganho de desempenho. Ademais, a GPU possui um pipeline gráfico, o que significa que a lógica das operações realizadas na mesma seguem o mesmo modelo de aplicações estritamente gráficas.

A estrutura de um programa em CUDA normalmente consiste em duas partes distintas: código a ser executado na CPU e código a ser executado na GPU. O princípio básico de um programa nesta estrutura é utilizar a CPU pra controlar o fluxo de execução do programa e a GPU para realizar as operações pesadas sobre os dados. A CPU utiliza um modelo SIMT (single instruction multiple threads) para realizar processamentos escaláveis, o que não requer a conversão de dados para vetores e também permite branching arbitrário de threads, e a GPU SIMD. Por isso, o programa sempre será inicializado a partir da CPU, em algum momento irá agregar os dados em blocos, passá-los como parâmetro para uma função da GPU, que será executada em SIMD e receber o retorno.

Um programa CUDA tem o seguinte fluxo básico:

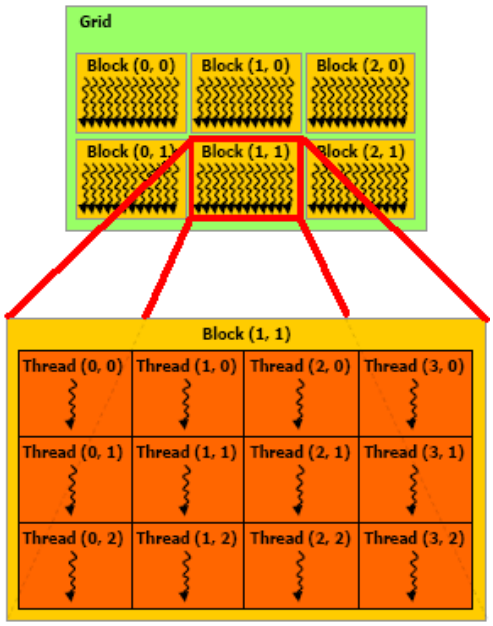
- O host inicializa um vetor com dados.
- O array é copiado da memória do host para a memória do dispositivo CUDA.
- O dispositivo CUDA opera sobre o vetor de dados.
- O array é copiado de volta para o host.





4. Estrutura de um Programa

A idéia da programação em CUDA consiste em definir funções em Linguagem C, chamadas de kernels, que quando chamados são executados N vezes em paralelo por N diferentes Threads de CUDA. A paralelização é realizada de forma automática.

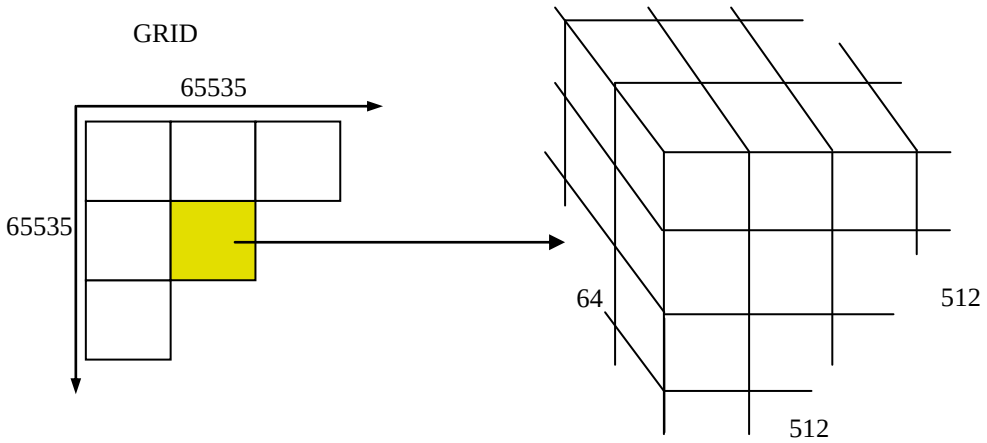


Uma thread é definida em duas partes: a declaração e a chamada. Par declarar deve-se utilizar o especificador **__global__**, e para a chamada deve-se informar o número de threads que deverão ser criadas, em uma sintaxe própria: <<<>>>. O **primeiro** argumento especifica a dimensão do grid (número de blocos), enquanto que o **segundo** a dimensão dos blocos <<<num_blocks, dim_blocks>>>. A cada bloco está associado um *thread block* (é o conjunto de threads do bloco), que pode ser 1D, 2D ou 3D, de forma que possam ser processados vetores, matrizes ou volumes. O número de threads de um bloco é igual ao número de elementos a serem processados no *Block*.

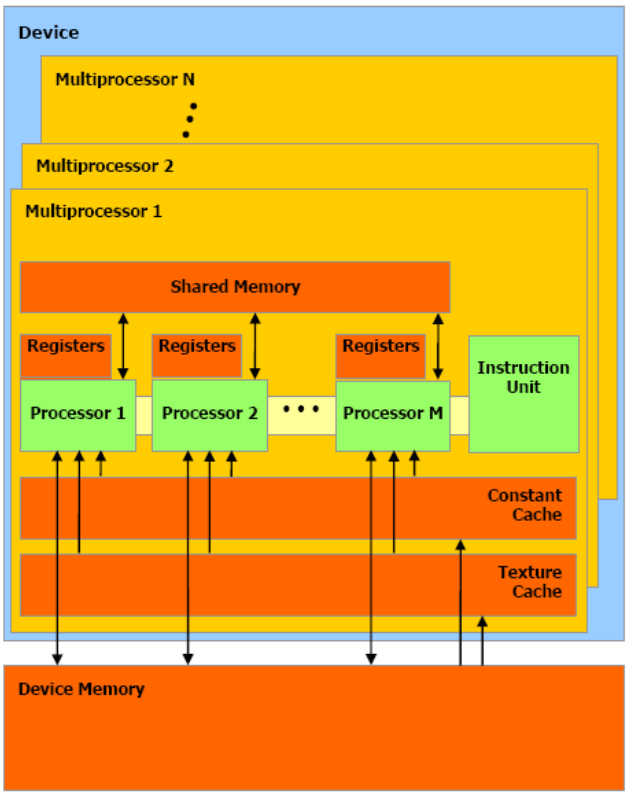
Para se definir a dimensão do bloco utiliza-se uma variável to tipo dim3, que pode ser inicializada com 1, 2 ou 3 parâmetros. Cada parâmetro indica o número de elementos em cada dimensão. Em dim3 dimBlock(N,N) está-se definindo uma matriz 2D com NxN elementos. Em dimBlock(N) define-se um vetor com N elementos.

O tamanho do grid e número de threads suportadas por cada elemento do grid dependem da capacidade de computação do dispositivo, conforme a seguinte tabela. Para o *Compute Capability 1.0*, tem-se as seguintes limitações nos parâmetros para invocação dos kernels.

- A dimensão máxima em x, y e z para um *thread block*, ou seja, a dimensão do bloco, é 512, 512 e 64 respectivamente. Isso quer dizer que para processar um vetor com 1000 elementos, deve-se ter pelo menos dois blocos <<<2, 500>>>. Se for feita chamada <<<1, 1000>>> nada será processado.
- O tamanho de cada dimensão do grid de *thread blocks* está limitada em 65535 e pode ser em uma ou duas dimensões. Desta forma pode-se ter um limite hipotético de 65535x65535 blocks x 512x512x64 threads = 72.055.395.031.449.600 elementos (72 Peta). Considerando-se a dimensão máxima do grid e do bloco, pode-se processar vetores unidimensionais com grids unidimensionais com até 65535x512 = 335.604.735 elementos.



Comente sobre o que são os multiprocessadores em relação a grids, blocos, memórias, etc.

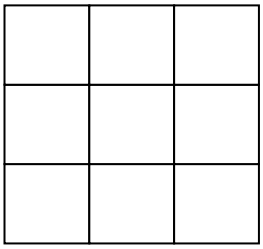



```
__global__ void vecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
int main()
{
    dim3 dimGrid(1);
    dim3 dimBlock(N);
    vecAdd<<<dimGrid, dimBlock>>>(A, B, C);
    vecAdd<<<1,       N       >>>(A, B, C);
}
```

1 bloco com dimensão 1(N)

Para processar um vetor com N elementos, em um único grupo, são criadas N threads, sendo cada uma associada a um Id do vetor a ser processado, de forma que o vetor possa ser processado em paralelo. Neste caso, ocorre um mapeamento direto do índice do vetor com o Id da thread. O identificador `__global__` é usado para indicar que a função (kernel) somente pode ser executado no device.

```
__global__ void matAdd(float A[N][N], float B[N][N], float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}
int main()
{
    dim3 dimBlock(N, N);
    matAdd<<<1, dimBlock>>>(A, B, C);
}
```

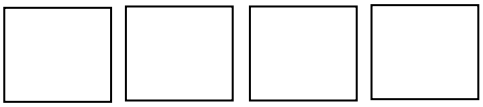


1 bloco com dimensão 2 (N,N)

Para processar um vetor com NxN elementos, em um único grupo, são criadas NxN threads, sendo cada uma associada a um Id (x,y) da matriz a ser processada, de forma que a matriz possa ser processada em paralelo.

Neste caso, dada uma matriz de tamanho (D_x, D_y), o Id da thread de índice (x,y) é dado por (x + y D_x). Neste exemplo são criadas NxN threads, ou seja, uma para casa elemento da matriz.

```
__global__ void vecAdd(float *a, int N)
{
  int idx = blockIdx.x*blockDim.x + threadIdx.x;
  if( idx < N )
     a[idx] = a[idx]+1.f;
}
int main()
{
   // Kernel invocation
   vecAdd<<<4, N/4>>>(A, N);
}
```



4 blocos com dimensão 1 (N/4)

Neste exemplo foram criados blocos com tamanho N/4 de forma arbitrária. Já o *grid* foi criado com um número de *blocks* suficientes para se ter uma thread para casa elemento do vetor a ser processado. Caso a relação dim_grid x dim_bloco não for perfeita (e.g. processar um vetor com 8 elementos com um grid de 3 blocos), deve-se utilizar um algoritmo simples para determinar o número de blocos.

```
  int block_size = 3;
  int n_blocks = N/block_size + (N%block_size == 0 ? 0:1);
  vecAdd <<< n_blocks, block_size >>> (A, N);
```

A mesma situação do exemplo anterior é apresentada no seguinte exemplo, com a diferença que agora está-se processando uma matriz.

```
__global__ void matAdd(float A[N][N], float B[N][N], float C[N][N])
{
   int i = blockIdx.x * blockDim.x + threadIdx.x;
   int j = blockIdx.y * blockDim.y + threadIdx.y;
   if (i < N && j < N)
       C[i][j] = A[i][j] + B[i][j];
}
int main()
{
   dim3 dimBlock(16, 16);
   dim3 dimGrid((N + dimBlock.x - 1) / dimBlock.x,
                (N + dimBlock.y - 1) / dimBlock.y);
   matAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

Antes de fazer a chamada de um kernel, deve-se fazer a inicialização da memória do host (CPU) e do device (GPU).

```
int main(void)
{
  cudaError_t t;

  float *a_h, *a_d;  // Pointer to host & device arrays
  const int N = 512;  // Number of elements in arrays
  size_t size = N * sizeof(float);

  a_h = (float *)malloc(size);        // Allocate array on host
  t = cudaMalloc((void **) &a_d, size);
  const char *error = cudaGetErrorString(t);
  printf("\nError: %s\n", error);

  init_vector(a_h, N);

  dim3 dimBlock(N);
  init_vector(a_h, N);
```

```
  cudaMemcpy(a_d, a_h, size, cudaMemcpyHostToDevice);
  incrementa_1 <<< 1, dimBlock >>> (a_d);
  cudaMemcpy(a_h, a_d, size, cudaMemcpyDeviceToHost);

  free(a_h);
  cudaFree(a_d);
  return 0;
}
```

5. Análise de desempenho

Para se ter uma idéia de desempenho, realizou-se vários testes sobre o seguinte algoritmo (código para execução em CPU), levando-se em consideração o número de blocos e threads por bloco.

```
const int N = 512;
#define LOOPS 100000
__host__ void incrementa_1_CPU(float *a, int N)
{
  float res;
  for(int idx=0; idx<N; idx++)
  {
     res = a[idx];
     for(int i=0; i<LOOPS; i++)
        res = sin((float)res) + cos((float)i) + sqrt( res+10.0 );
     a[idx] = res;
  }
}
```

O seguinte código foi usado em GPU, considerando-se a existência de um único bloco com N threads.

```
__global__ void incrementa_1(float *a)
{
  int idx = threadIdx.x;
  float res;
  res = a[idx];
  for(int i=0; i<LOOPS; i++)
        res = sin((float)res) + cos((float)i) + sqrt( res+10.0 );
  a[idx] = res;
}
```

Para testes com vários blocos, utilizou-se o seguinte algoritmo

```
__global__ void incrementa_N(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  float res = a[idx];
  if (idx<N)
  {
     for(int i=0; i<LOOPS; i++)
       res = sin((float)res) + cos((float)i) + sqrt( res+10.0 );
     a[idx] = res;
  }
}
```

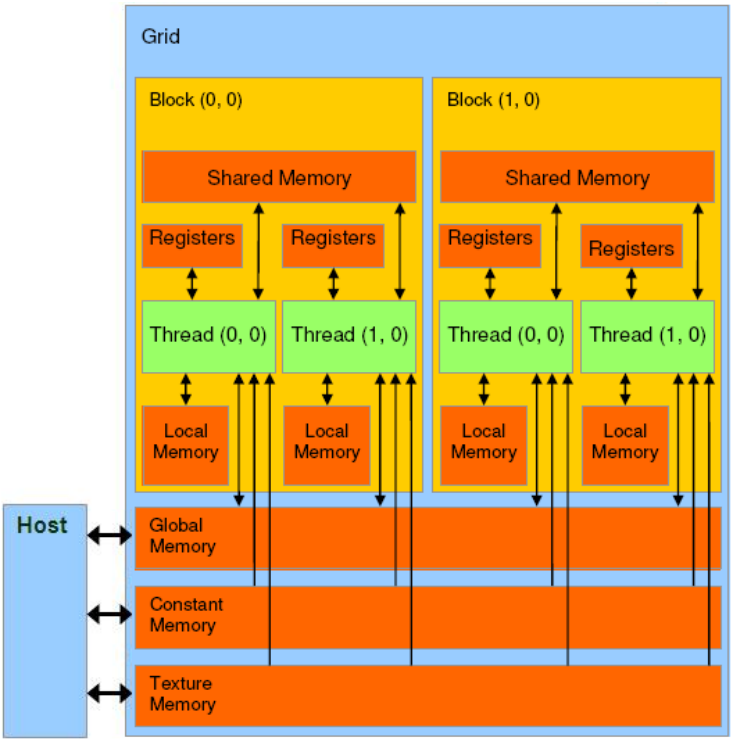
A seguinte tabela ilustra o tempo de processamento em cada teste realizado

Formato dos dados	Tempo	Razão c/ melhor tempo
`<<<1, N>>> c/ incrementa_1`	609 ms	2.38
`<<<1, N>>> com res = a[idx]`	607 ms	2.38
`<<<N,1>>>`	1.643 ms	6.44
`<<<N/2,2>>>`	936 ms	3.67
`<<<N/4,4>>>`	590 ms	2.31
`<<<N/8,8>>>`	277 ms	1.08
`<<<N/16, 16>>> = <<<32, 16>>>`	256 ms	1.0
`<<<N/32, 32>>> = <<<16, 32>>>`	255 ms	1.0
`<<<N/64, 64>>> = <<<8, 64>>>`	256 ms	1.0
`<<<N/128,128>>> = <<<4, 128>>>`	271 ms	1.06
`<<<N/256,256>>> = <<<2, 256>>>`	336 ms	1.31
`<<<1, N>>> c/ incrementa_N`	611 ms	2.39
`CPU C2Q 2.4Ghz single thread`	6.294 ms	24.68

6. Gerenciamento de Memória

Uma thread pode acessar memória de vários locais distintos: (comente tudo isso)

1. Da memória local da thread;
2. Da memória compartilhada do *block* – esta memória pode ser acessada por qualquer thread do *block*;
3. Memória de textura;
4. Da memória global – pode ser acessada por qualquer thread do *grid*; ou
5. Memória constante (leitura).



Exemplo da multiplicação de matrizes: compare o desempenho usando e não o __shared__. Esqueci de mencionar este exemplo na aula de hoje.

Shared Memory (pg. 60 tutorial cuda)

Because it is on-chip, the shared memory space is much faster than the local and global memory spaces. In fact, for all threads of a warp, accessing the shared memory is as fast as accessing a register as long as there are no bank conflicts between the threads, as detailed below.

7. Configurando esquema de cores para o Visual Studio

Como o código CUDA deve ser escrito em arquivos com extensão .cu, este não é reconhecido pelo Visual Studio, não habilitando assim o colorimento de palavras chave da linguagem, que em sua grande maioria são iguais à linguagem C. Para habilitar o colorimento, siga os seguintes passos:

1. Abra o editor de registro (regedit);
2. Adicione uma chave para extensão .cu, para subchave [HKEY_LOCAL_MACHINE\ SOFTWARE\ Microsoft\VisualStudio\7.1\Languages\File Extensions]. Ao final deverá aparecer um novo ícone;
3. Copie a informação da extensão **.c** para essa nova extensão, por meio da opção *modify*.

Referencias

http://llpanorama.wordpress.com/cuda-tutorial/
http://www.nvidia.com/object/cuda_home.html#