

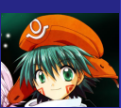
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Utilização da unidade processamento gráfico para propósito geral (GPGPU)

Rafael Guedes Lang
Anderson Gonçalves Marco

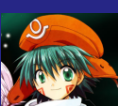
12 de abril de 2011



Sumário

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco



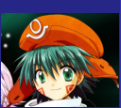
o que é (GPGPU) ?

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

E a utilização da placa de vídeo como um co-processador matemático.

Através criação de rotinas que rodam dentro do processador da placa de vídeo(GPU),por meio de bibliotecas como: Brook, Close to Metal(ATI), Sh, OpenCL(Apple), CUDA(NVIDIA)



Historia das placas de vídeo 3D

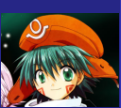
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Foram criadas inicialmente para aumentar a qualidade gráfica dos jogos (Hoje qualquer jogo 3D precisa de uma placa 3D).

Antes de 1999 as placa aceleradoras 3D, domesticas, cuidavam basicamente da aplicação de texturas em superfícies de polígonos. Com a Geforce 256 as placas de vídeo passaram a incorporar outras funções, como transformações de vértices, que antes eram desempenhadas pela FPU do processador central do computador.

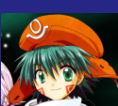
Para isto o processador da placa de vídeo precisou evoluir para uma GPU(Unidade de processamento gráfico), tornando-se extremamente eficaz em operações com ponto flutuante.



Sumário

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)



Rafael Guedes
Lang
Anderson
Gonçalves
Marco

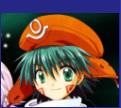


Comparação entre Processadores Gráficos e Processadores Centrais

Utilização da unidade processamento gráfico para propósito geral (GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

GPUs	CPUs
	
Propósito específico (lidar com floats)	Propósito genérico
Unidades de processamento simples e em grande quantidade	Unidades processamento complexas e em pequena quantidade
Ruim em predição de desvio (if,else)	Bom em predição de desvio
Não segue completamente o padrão para pontos flutuantes, IEEE 754	Segue o padrão IEEE 754
Paralelismo agressivo	Paralelismo modesto



Comparação entre Processadores Gráficos e Processadores Centrais

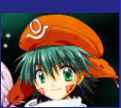
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Estas diferenças conferem as GPUs um ganho brutal em operações aritméticas de floats.

O Gráfico do slide seguinte confirma isto.

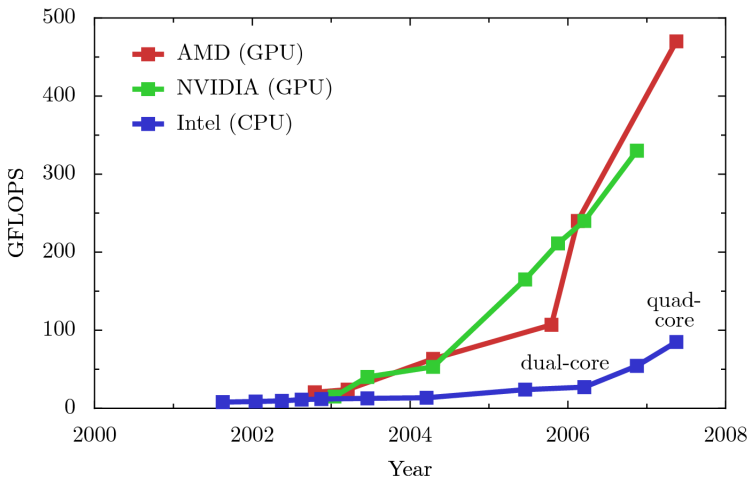
Este gráfico está um pouco desatualizada com relação a o desempenho das GPUs, a nova GPU GTX280 da Nvidia alcança a marca de 1 Teraflop

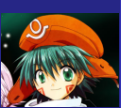


Comparação do crescimento dos Gflops, ao longo do tempo, entra as GPUs Nvidia, ATI e CPUs Intel

Utilização da unidade processamento gráfico para propósito geral (GPGPU)

Rafael Guedes Lang
Anderson Gonçalves
Marco

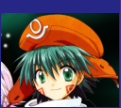




Sumário

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco



Tipos de processadores da GPU

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

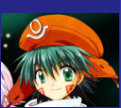
Processador Central

Responsavel por gerenciar as tarefas mais comuns da placa de vídeo como:

- Controlar a saída de vídeo.
- Passar uma imagem de 3D para 2D, para que a mesma possa ser exibida no monitor.
- Aceleração de vídeo 2D.

Stream Processor(Blocos)

Basicamente é um cluster onde ficam os Scalar Processor e a Memória Compartilhada (a ser vista mais a frente).



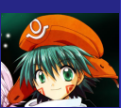
Tipos de processadores da GPU

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Scalar Processor

Responsavel por processar códigos GPGPU e instruções de Shaders.



Tipos de Memórias da GPU

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

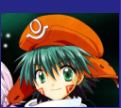
Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Registradores

Cada Stream Processor possui 8192 registradores a serem divididos entre os Scalar Processors do Stream Processor. A thread de um Scalar Processor não enxerga os registradores das outras threads rodando na GPU.

Memória Compartilhada

Cada Stream Processor possui 16kb de memória, que é quase tão rápida quanto os registradores, esta memória é visível a todos os Scalar Processors de um Stream Processor, mas um Scalar Processor de um Stream Processor não pode ver a memória compartilhada de outro Stream Processor.



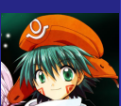
Tipos de Memórias da GPU

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Memória Global (RAM)

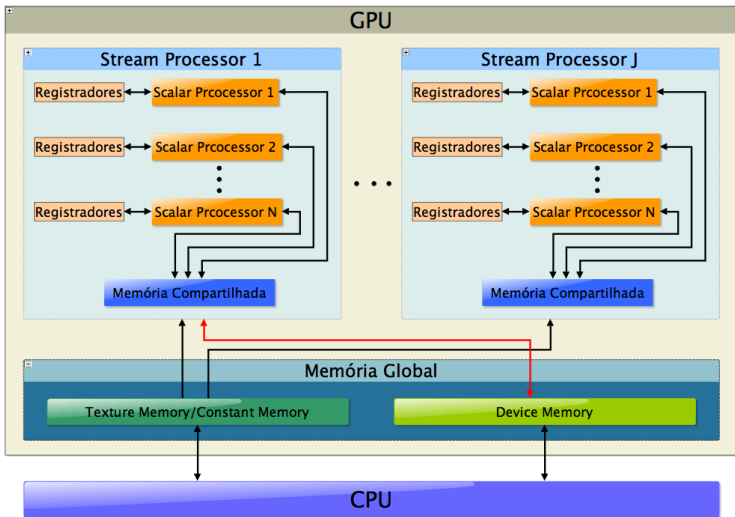
Visível por todas as threads de uma GPU é onde esta a maior parte da memória da GPU, muito mais lenta que registradores ou memória compartilhada, é a única memória que a CPU pode ler e escrever. Pode ser alocada pela CPU, e só por ela, como device memory (a CPU e todas as threads da GPU podem ler e escrever nela) ou texture memory (a CPU pode escrever e as threads da GPU só podem ler) sendo que esta última permite um acesso mais rápido a memória global.

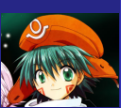


Esquema da arquitetura de GPUs

Utilização da unidade processamento gráfico para propósito geral (GPGPU)

Rafael Guedes Lang Anderson Gonçalves Marco





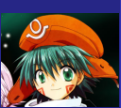
Arquitetura NVIDIA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

A arquitetura das placas NVIDIA pode ser encarada como sendo um caso especial de arquitetura de GPU, cada bloco mencionado anteriormente para as placas NVIDIA é um processador SM (Stream Processor), cada processador SM possui 8 processadores SP (Scalar Processor), é cada SP, através de um pipeline agressivo, pode executar até 4 instruções por clock, deste modo o número máximo de tarefas sendo executadas simultaneamente numa GPU nvidia é:

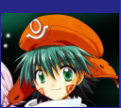
$8 * 4 * \text{numero_de_processadores_SM}$



Sumário

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

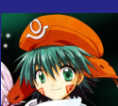


o que é CUDA?

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- É uma biblioteca e um compilador para criação de rotinas para GPUs Nvidia.

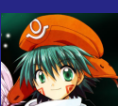


o que é CUDA?

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- É uma biblioteca e um compilador para criação de rotinas para GPUs Nvidia.
- Esta API está num nível intermediário entre as APIs de baixo nível (Cg Toolkit) e alto nível (Brooks), sendo necessário ainda algum conhecimento de como a GPU funciona internamente para poder utilizá-la adequadamente.

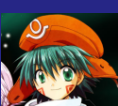


o que e CUDA?

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- E uma biblioteca e um compilador para criação de rotinas para GPUs Nvidia.
- Esta API esta num nível intermediário entre as APIs de baixo nível (Cg Toolkit) e alto nível (Brooks), sendo necessário ainda algum conhecimento de como a GPU funciona internamente para poder utiliza-la adequadamente.
- Entretanto ela e uma API estável, e vem sendo utilizada em muitos projetos científicos.

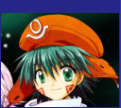


o que e CUDA?

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- E uma biblioteca e um compilador para criação de rotinas para GPUs Nvidia.
- Esta API esta num nível intermediário entre as APIs de baixo nível (Cg Toolkit) e alto nível (Brooks), sendo necessário ainda algum conhecimento de como a GPU funciona internamente para poder utiliza-la adequadamente.
- Entretanto ela e uma API estável, e vem sendo utilizada em muitos projetos científicos.
- Possui uma grande comunidade e boa documentação.



Requisitos para instalar CUDA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Uma placa de vídeo com GPU G80 (Gerfoce 8, Quadro) ou posterior.

Ter instalado no computador o driver da placa de vídeo, recomenda-se as ultimas versões do driver.



Compilando programas em CUDA

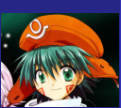
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Para se compilar um programa em CUDA deve-se usar o comando `nvcc arquivo-fonte.cu -o arquivo-de-saida`.

Não e necessário uma placa Nvidia para se compilar, mais e necessário para se rodar o executável.

Os arquivos fontes CUDA tem a extensão `*.cu`.

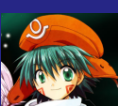


Primeiro programa em CUDA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

No próximo slide será visto um simples programa, o que ele faz e pegar um vetor elevar todas as componentes deste vetor ao quadrado e multiplicar por um escalar.



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

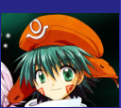
```
1 // incrementArray.cu
2 #include <assert.h>
3 #include <cuda.h>
4 #include <stdio.h>
5 __global__ void mult_vet(float *vet,float *vetr,float mul)
6 {
7     int idx = blockIdx.x * blockDim.x + threadIdx.x;
8     vetr[idx]=vet[idx]*vet[idx]*mul;
9 }
10
11
12 int main(void)
13 {
14     float *a_h, *b_h;           // pointers to host memory
15     float *a_d,*b_d;           // pointer to device memory
16     int i, N = 256;
17     size_t size = N*sizeof(float);
18     // allocate arrays on host
19     a_h = (float *)malloc(size);
20     b_h = (float *)malloc(size);
21     // allocate array on device
22     cudaMalloc((void **) &a_d, size);
23     cudaMalloc((void **) &b_d, size);
24     // initialization of host data
25     for (i=0; i<N; i++) a_h[i] = (float)45;
26     for (i=0; i<N; i++) b_h[i] = (float)0;
27     a_h[251]=56.0f;
28     // copy data from host to device
29     cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
30     cudaMemcpy(b_d, b_h, sizeof(float)*N, cudaMemcpyHostToDevice);
31     int blockSize = 16;
```



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

```
32     int nBlocks = 16;
33     mult_vet <<< nBlocks, blockSize >>> (a_d,b_d,32.0f);
34     cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
35     for (i=0; i < N; i++) printf("%f \n",b_h[i] );
36     free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
37
38 }
```

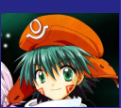


Explicando o código acima

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- Os vetores (ponteiros) `a_h` e `b_h` são alocados na memória do host (**memoria ram do computador**) e inicializados com um valor qualquer (**linhas 20, 21, 26 e 27**).
- Os vetores (ponteiros) `a_d` e `b_d` são alocados na memory device da GPU (**linhas 23, 24**), o conteúdo de `a_h` e `b_h` e copiado para `a_d` e `b_d` (**linhas 30, 31**).
- define-se e a quantidade de processos a serem executados na GPU (**linhas 32, 33**). O ideal seria que `block_Size` fosse um multiplo de 64 e `nBlocks` fosse ≥ 64 (a Nvidia recomenda ≥ 100), pois neste arranjo é que se obtém o máximo de performance.



Explicando o código acima

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- Chama-se a função a ser executada na GPU.
- Copia-se valor do vetor `b_d` para `b_h` (linha 35).
- Desaloca-se os vetores (linha 37).

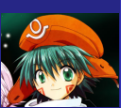


Aumentando o desempenho

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- Repare que no vetor `a_d` os processadores da GPU fazem apenas operações de leitura não de escrita.

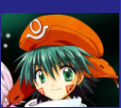


Aumentando o desempenho

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- Repare que no vetor `a_d` os processadores da GPU fazem apenas operações de leitura não de escrita.
- Colocar este vetor então na memory texture ou na memory constant, aumenta a velocidade de transferência, aumentando o desempenho.

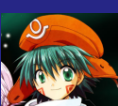


Aumentando o desempenho

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

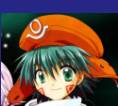
- Repare que no vetor `a_d` os processadores da GPU fazem apenas operações de leitura não de escrita.
- Colocar este vetor então na memory texture ou na memory constant, aumenta a velocidade de transferência, aumentando o desempenho.
- No próximo slide uma modificação usando memory texture.



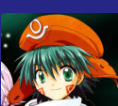
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

```
1  #include <assert.h>
2  #include <cuda.h>
3  #include <stdio.h>
4  texture<float, 1, cudaReadModeElementType> texRef;//novo
5  __global__ void mult_vet(float *vetr,float mul)
6  {
7      int idx = blockIdx.x * blockDim.x + threadIdx.x;//ID da Thread
8      vetr[idx]= tex1Dfetch(texRef, idx)*tex1Dfetch(texRef, idx)*mul;//novo
9  }
10
11
12  int main(void)
13  {
14      float *a_h, *b_h;           // pointers to host memory
15      float *a_d,*b_d;           // pointer to device memory
16      int i, N = 256;
17      size_t size = N*sizeof(float);
18      // allocate arrays on host
19      a_h = (float *)malloc(size);
20      b_h = (float *)malloc(size);
21      // allocate array on device
22      cudaMalloc((void **) &a_d, size);
23      cudaMalloc((void **) &b_d, size);
24      // initialization of host data
25      for (i=0; i<N; i++) a_h[i] = (float)45;
26      for (i=0; i<N; i++) b_h[i] = (float)0;
27      a_h[251]=56.0f;
28      // copy data from host to device
29      cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
30      cudaMemcpy(b_d, b_h, sizeof(float)*N, cudaMemcpyHostToDevice);
31      cudaBindTexture(0,texRef, a_d, sizeof(float)*N);//novo
```



```
32     int blockSize = 16;
33     int nBlocks = 16;
34     mult_vet <<< nBlocks, blockSize >>> (b_d,32.0f);
35     cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
36     for (i=0; i < N; i++) printf("%f \n",b_h[i] );
37     cudaUnbindTexture(texRef);//novo
38     free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
39 }
40
```



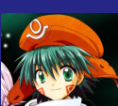
Caso 1D

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Explicando o código visto anteriormente

- Na linha 4 é criada um ponteiro para uma região da textura.
- Na linha 31 copia-se o conteúdo do vetor `a_d` para uma região da memory texture.
- Na linha 7 se acessa uma região da textura multiplica-se o valor desta região por ele mesmo e pela variável `mul`.
- Na linha 37 desaloca-se a região de textura para onde o vetor `a_d` foi copiado.



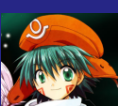
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Outras maneiras de usar a memória de textura

- Alocar o vetor na estrutura de array do CUDA, para então passar esta estrutura para a memória de texturas, permite que a GPU realize algumas operações de maneira "automática" sobre os dados deste array (como mapeamento do discreto para contínuo nos índices, dos elementos, e interpolação dos pontos com os seus elementos vizinhos) que para alguns problemas, com que se trabalha na computação científica, podem ser úteis.
- Alocar a memória em 2 dimensões, isto melhora o desempenho quando se trabalha com matrizes.

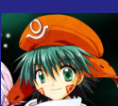
No próximo slide um exemplo, de como usar a estrutura de array do CUDA.



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

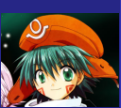
```
1  #include <assert.h>
2  #include <cuda.h>
3  #include <stdio.h>
4  texture<float, 1, cudaReadModeElementType> texRef;//novo
5  __global__ void mult_vet(float *vetr,float mul)
6  {
7      int idx = blockIdx.x * blockDim.x + threadIdx.x;//ID da Thread
8      vetr[idx]= tex1D(texRef,idx)*tex1D(texRef,idx)*mul;//novo
9  }
10 int main(void)
11 {
12     float *a_h, *b_h;           // pointers to host memory
13     float *b_d;                // pointer to device memory
14     int i, N = 256;
15     size_t size = N*sizeof(float);
16     // allocate arrays on host
17     a_h = (float *)malloc(size);
18     b_h = (float *)malloc(size);
19     // allocate array on device
20     cudaMalloc((void **) &b_d, size);
21     // initialization of host data
22     for (i=0; i<N; i++) a_h[i] = (float)i;
23     for (i=0; i<N; i++) b_h[i] = (float)0;
24     a_h[251]=56.0f;
25     // copy data from host to device
26     cudaArray *texArray = 0;
27     cudaChannelFormatDesc cf = cudaCreateChannelDesc<float>();
28     cudaMallocArray(&texArray, &cf,256,1);
29     cudaMemcpyToArray(texArray, 0,0, a_h, size, cudaMemcpyHostToDevice);
30     texRef.normalized = 0;
31     texRef.filterMode = cudaFilterModePoint;
```



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

```
32     texRef.addressMode[0] = cudaAddressModeClamp;
33     cudaBindTextureToArray(texRef, texArray);
34     int blockSize = 16;
35     int nBlocks = 16;
36     mult_vet <<< nBlocks, blockSize >>> (b_d,32.0f);
37     cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
38     for (i=0; i < N; i++) printf("%f \n",b_h[i] );
39     cudaUnbindTexture(texRef);//novo
40     free(a_h); free(b_h);  cudaFree(b_d);cudaFreeArray(texArray);
41 }
42
```



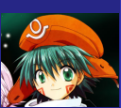
Caso 1D

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Explicando o código visto anteriormente

- Na linha 4 é criada um ponteiro para uma região da textura.
- Na linha 29 aloca-se espaço para um array CUDA.
- Na linha 30 copia-se o conteúdo do vetor `a_h` para um array CUDA.
- Na linha 31 diz como a região de textura deve ser mapeada, 0 para ser mapeada da forma normal, onde índices vão de 0 ao tamanho do vetor-1, diferente de 0 para ser mapeada na forma contínua, onde os índices vão de 0.0 a 1.0.
- Na linha 32, diz se a região de textura vai (`cudaFilterModeLinear`) ou não (`cudaFilterModePoint`) interpolar os valores passados a ela.



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

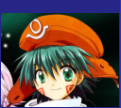
Cont ...

- Na linha 33 diz como a região de textura vai lidar com índices que extrapolam as dimensões do vetor (ou da matriz), colocadas em sua região.
- Na linha 34 copia-se o conteúdo do array CUDA para a região de textura;
- Na linha 8 se acessa uma região da textura multiplica-se o valor desta região por ele mesmo e pela variável `mul`.
- Na linha 40 desaloca-se a região de textura para onde o vetor `a_d` foi copiado.
- Na linha 41 desaloca-se as variáveis alocadas dinamicamente.



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

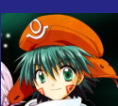


Caso 2D

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

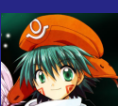
No próximo slide será visto um exemplo de como alocar matrizes na memória de textura.



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

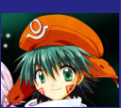
```
1  #include <assert.h>
2  #include <cuda.h>
3  #include <stdio.h>
4  texture<float, 2, cudaReadModeElementType> texRef;//novo
5  __global__ void mult_vet(float *vetr,float mul)
6  {
7      int idx = blockIdx.x * blockDim.x + threadIdx.x;//ID da Thread
8      // vetr[idx]= tex2D(texRef,threadIdx.x,blockIdx.x)*tex2D(texRef, threadIdx.x,blockIdx.x)*mul
9      vetr[idx]=tex2D(texRef, threadIdx.x,blockIdx.x);//novo
10 }
11
12
13 int main(void)
14 {
15     float *a_h, *b_h;           // pointers to host memory
16     float *b_d;                // pointer to device memory
17     int i, N = 256;
18     size_t size = N*sizeof(float);
19     // allocate arrays on host
20     a_h = (float *)malloc(size);
21     b_h = (float *)malloc(size);
22     // allocate array on device
23     cudaMalloc((void **) &b_d, size);
24     // initialization of host data
25     for (i=0; i<N; i++) a_h[i] = (float)i;
26     for (i=0; i<N; i++) b_h[i] = (float)0;
27     a_h[251]=56.0f;
28     // copy data from host to device
29     cudaArray *texArray = 0;
30     cudaChannelFormatDesc cf = cudaCreateChannelDesc<float>();
31     cudaMallocArray(&texArray, &cf, 16, 16);
```



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

```
32     cudaMemcpyToArray(texArray, 0,0, a_h, size, cudaMemcpyHostToDevice);
33     texRef.normalized = 0;
34     texRef.filterMode = cudaFilterModePoint;
35     texRef.addressMode[0] = cudaAddressModeClamp;
36     texRef.addressMode[1] = cudaAddressModeClamp;
37     // texRef.addressMode[1] =cudaAddressModeWrap ;
38
39     cudaBindTextureToArray(texRef, texArray);
40     int blockSize = 16;
41     int nBlocks = 16;
42     mult_vet <<< nBlocks, blockSize >>> (b_d,32.0f);
43     cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
44     for (i=0; i < N; i++) printf("%f \n",b_h[i] );
45     cudaUnbindTexture(texRef);//novo
46     free(a_h); free(b_h);  cudaFree(b_d);cudaFreeArray(texArray);
47 }
48
```



Constant Memory

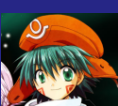
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

É uma memória rápida vista por todos os processadores da GPU mas apenas o host pode escrever nela.

Limitações da Constant Memory

Ela é uma memória de alocação estática e atribuição estática.



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

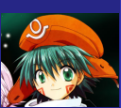
```
1  #include <assert.h>
2  #include <cuda.h>
3  #include <stdio.h>
4  __constant__ float a_c[255]={45.0f,70.0f};
5  __global__ void mult_vet(float *vetr,float mul)
6  {
7      int idx = blockIdx.x * blockDim.x + threadIdx.x;//ID da Thread
8      vetr[idx]= a_c[idx]*a_c[idx]*mul;//novo
9  }
10
11
12 int main(void)
13 {
14     float *b_h;           // pointers to host memory
15     float *b_d;           // pointer to device memory
16     int i, N = 256;
17     size_t size = N*sizeof(float);
18     // allocate arrays on host
19     b_h = (float *)malloc(size);
20     // allocate array on device
21     cudaMalloc((void **) &b_d, size);
22     // initialization of host data
23     for (i=0; i<N; i++) b_h[i] = (float)0;
24     // copy data from host to device
25     int blockSize = 16;
26     int nBlocks = 16;
27     mult_vet <<< nBlocks, blockSize >>> (b_d,32.0f);
28     cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
29     for (i=0; i < N; i++) printf("%f \n",b_h[i] );
30     free(b_h); cudaFree(b_d);
31 }
```



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

32



Comunicação entre processos

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Como CUDA segue o modelo de programação Memória Compartilha a comunicação entre processos se da através da memória.

Para este fim usa-se a device memory e a shared memory.

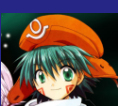
Device memory

E mais lenta usa-se para se fazer comunicação entre processos de diferentes blocos e funções da GPU.

Shared Memory

E mais rápida, usa-se para se fazer comunicação entre diferentes processos de um mesmo bloco.

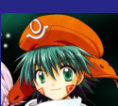
No próximo slide um exemplo de comunicação utilizando a device memory.



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

```
1  #include <assert.h>
2  #include <cuda.h>
3  #include <stdio.h>
4  texture<float, 1, cudaReadModeElementType> texRef;
5  __device__ float dev[256]; // novo
6  __global__ void mult_vet(float *vetr, float mul)
7  {
8      int idx = blockIdx.x * blockDim.x + threadIdx.x; // ID da Thread
9      vetr[idx] = tex1Dfetch(texRef, idx) * tex1Dfetch(texRef, idx) * mul; // novo
10     dev[idx] = vetr[idx] * 2.0f;
11 }
12
13 __global__ void mult_vet2(float *vetr) {
14     int idx = blockIdx.x * blockDim.x + threadIdx.x; // ID da Thread
15     vetr[idx] = dev[idx] * vetr[idx];
16 }
17
18 __global__ void mult_vet(float *vetr, float mul);
19 int main(void)
20 {
21     float *a_h, *b_h; // pointers to host memory
22     float *a_d, *b_d, *b_d2; // pointer to device memory
23     int i, N = 256;
24     size_t size = N * sizeof(float);
25     // allocate arrays on host
26     a_h = (float *) malloc(size);
27     b_h = (float *) malloc(size);
28     // allocate array on device
29     cudaMalloc((void **) &a_d, size);
30     cudaMalloc((void **) &b_d, size);
31     cudaMalloc((void **) &b_d2, size);
```



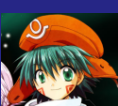
```
32 // initialization of host data
33 for (i=0; i<N; i++) a_h[i] = 7.3242f;
34 for (i=0; i<N; i++) b_h[i] = (float)0;
35 a_h[251]=6.0f;
36 // copy data from host to device
37 cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
38 cudaMemcpy(b_d, b_h, sizeof(float)*N, cudaMemcpyHostToDevice);
39 cudaBindTexture(0,texRef, a_d, sizeof(float)*N);//novo
40 int blockSize = 16;
41 int nBlocks = 16;
42 mult_vet <<< nBlocks, blockSize >>> (b_d,2.234f);
43 cudaThreadSynchronize();
44 mult_vet2 <<< nBlocks, blockSize >>>(b_d);
45 cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
46 for (i=0; i < N; i++) printf("%f \n",b_h[i] );
47 cudaUnbindTexture(texRef);//novo
48 free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
49 }
50
51
```



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

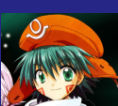
No próximo slide um exemplo de comunicação utilizando a shared memory.



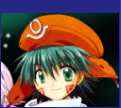
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

```
1  #include <assert.h>
2  #include <cuda.h>
3  #include <stdio.h>
4  texture<float, 1, cudaReadModeElementType> texRef;//novo
5  __global__ void mult_vet(float *vetr,float mul)
6  {
7      int idx = blockIdx.x * blockDim.x + threadIdx.x;//ID da Thread
8      __shared__ float shared;
9      if(threadIdx.x==0) shared=(float)blockIdx.x;
10     __syncthreads(); //sincroniza as threads dos blocos
11     vetr[idx]= tex1Dfetch(texRef, idx)*tex1Dfetch(texRef, idx)*mul*shared;//novo
12 }
13
14
15 __global__ void mult_vet(float *vetr,float mul);
16 int main(void)
17 {
18     float *a_h, *b_h;           // pointers to host memory
19     float *a_d,*b_d;           // pointer to device memory
20     int i, N = 256;
21     size_t size = N*sizeof(float);
22     // allocate arrays on host
23     a_h = (float *)malloc(size);
24     b_h = (float *)malloc(size);
25     // allocate array on device
26     cudaMalloc((void **) &a_d, size);
27     cudaMalloc((void **) &b_d, size);
28     // initialization of host data
29     for (i=0; i<N; i++) a_h[i] = (float)45;
30     for (i=0; i<N; i++) b_h[i] = (float)0;
31     a_h[251]=56.0f;
```



```
32 // copy data from host to device
33 cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
34 cudaBindTexture(0, texRef, a_d, sizeof(float)*N); //novo
35 int blockSize = 16;
36 int nBlocks = 16;
37 mult_vet <<< nBlocks, blockSize >>> (b_d, 32.0f);
38 cudaMemcpy(b_h, b_d, sizeof(float)*N, cudaMemcpyDeviceToHost);
39 for (i=0; i < N; i++) printf("%f \n", b_h[i] );
40 cudaUnbindTexture(texRef); //novo
41 free(a_h); free(b_h); cudaFree(a_d); cudaFree(b_d);
42 }
43
44
```



Tipos de rotinas

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

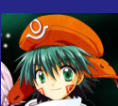
Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Rotinas bloqueantes

São aquelas que esperam que todas as chamadas CUDA antes dela tenham sido processadas para então poderem ser chamadas e quando chamadas ocupam todo processamento, só liberando o processador quando terminada sua execução.

Exemplos:

- `cudaMemcpy`.
- `cudaThreadSynchronize()` utilizada para sincronização.



Tipos de rotinas

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

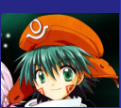
Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Rotinas não bloqueantes

Permitem a execução de outras chamadas CUDA, não bloqueantes, paralelamente.

Exemplos:

- `cudaMemcpyAsync`.
- As rotinas definidas, por nos, que rodam na GPU.



Estruturas de dados

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

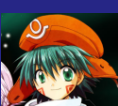
Rafael Guedes
Lang
Anderson
Gonçalves
Marco

CUDA possui estruturas de dados já prontas e permite a criação de novas estruturas de dados.

Estruturas de dados já definidas pelo CUDA

`char1, uchar1, char2, uchar2, char3, uchar3, char4, uchar4,`
`short1, ushort1, short2, ushort2, short3, ushort3, short4,`
`ushort4, int1, uint1, int2, uint2, int3, uint3, int4, uint4, long1,`
`ulong1, long2, ulong2, long3, ulong3, long4, ulong4, float1,`
`float2, float3, float4`

São vetores onde a parte em verde e o tipo de dado e a parte em vermelho e o numero de componentes $[x(1),y(2),z(3),w(4)]$.



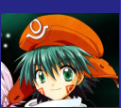
Estruturas de dados

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Exemplo

```
float4 posi;  
posi.x=3.0f;  
posi.y=10.0f;  
posi.z=2.0f;  
posi.w=-2.34f;
```



Criando estruturas de dados

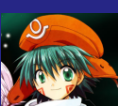
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Cria-se da mesma maneira que no c, mas deve-se fazer um alinhamento dos elementos da struct isto e feito com a diretiva `__align__(8)` (caso a estrutura tenha menos de 4 elementos) ou `__align__(16)` (caso a estrutura tenha 4 ou mais elementos), o motivo de se fazer isto e que assim é obtido um maior desempenho.

Exemplo de criação de uma struct com 2 elementos

```
typedef struct __align__(8){  
    float x;  
    float y;  
}vet;
```



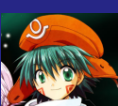
Cont...

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Exemplo de criação de uma struct com 5 elementos

```
typedef struct __align__(16){  
    float a;  
    float b;  
    float c;  
    float d;  
    float e;  
}vet2;
```



Funções matemáticas otimizadas para GPU

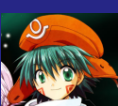
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

As funções matemáticas, otimizadas para floats, são as mesmas encontradas na biblioteca `math.h` do C-ANSI, porem elas devem precedidas por `__`.

Exemplo:

CPU	GPU
<code>sinf(float x)</code>	<code>__sinf(float x)</code>
<code>cosf(float x)</code>	<code>__cosf(float x)</code>
<code>logf(float x)</code>	<code>__logf(float x)</code>



Funções matemáticas otimizadas para GPU

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

O uso da diretiva `-use_fast_math` na hora da compilação, transforma as funções não otimizadas(sem `__`) em otimizadas.

Exemplo:

```
nvcc arquivo-fonte.cu -use_fast_math -o arquivo-de-saida
```

Problemas das funções otimizadas

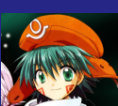
Elas possuem uma precisão menor que as não otimizadas.



Sumário

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

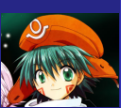


Historia

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Antes de 2010 não era possível debugar o código que se rodava na GPU, para se debugar o código era necessário fazer o mesmo rodar na CPU usando o parâmetro `-device-emulation` na hora da compilação do código, este parâmetro faz com que as funções que rodam na GPU passem a rodar na CPU e também faz o código ficar sequencial. Era comum então o código que se estava debugando não apresentar erros, e na hora de rodar na GPU o mesmo código rodar de maneira incorreta. Em 2010 a NVIDIA decidiu então investir no área de debuggers para CUDA. Dois projetos emergiram o CUDA-gdb (para linux) e o Parallel Nsight (para windows).



Debuggers CUDA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

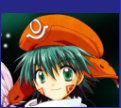
Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Parallel Nsight

Um debuger que se integra totalmente a IDE Visual Studio da Microsoft ele e todo wizard e é bem simples de usar, não apenas debuga código CUDA como também código de Shader (DirectX e HLSL) e OpenCL. Pode-se debugar o código que roda na GPU de maneira remota.

CUDA-gdb

Criado para o linux, a maneira de usar é muito parecida com a o gdb, roda em modo texto e por esta razão não é tão intuitivo quando o Parallel Nsight entretanto por ser baseado no gdb a curva de aprendizado se torna pequena para aqueles que sabem usar o gdb.

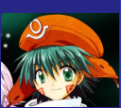


Debuggers CUDA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Nesta apresentação será mostrado como usar o CUDA-gdb somente.



Relembrando o gdb

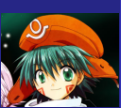
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Como dito anteriormente o CUDA-gdb é muito parecido com o gdb, será feita então uma rápida revisão sobre os comandos básicos do gdb.

Comandos gdb executados no shell do SO

- `gcc -g códigoFonte.c -o saída` → compila um arquivo c colocando o executável no arquivo saída e permite que este executável seja debugado pelo gdb (a opção `vESC -g` faz isto).
- `gdb arquivoexecutávelProntoParaDepuracao` → abre o gdb e diz que o arquivo executável que ele ira abrir para depurar é `arquivoexecutávelProntoParaDepuracao`.



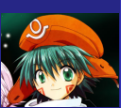
Relembrando o gdb

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Comandos gdb relativos a execução do programa*

- **run** *arg1 arg2 ... argn* ou **r** *arg1 arg2 ... argn* - *i* inicia o programa *o* com os *argn* argumentos de entrada passados para ele.
- **next** ou **n** → quando o programa esta rodando e chegou num ponto de parada este comando vai para o próxima linha do programa chegando na próxima linha o programa para novamente.
- **continue** ou **c** → quando o programa chega num ponto de parada este comando faz o programa continuar ate chegar no próximo ponto de parada.
- **bt** ou **backtrace** → diz quantas funções estão empilhadas na pilha de funções



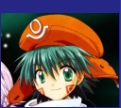
Relembrando o gdb

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Comandos gdb relativos a execução do programa*

- **up** e **down** → Sobe (comando up) ou desce (comando down) na pilha de funções.



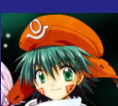
Relembrando o gdb

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Comandos gdb relativos a visualização de dados estáticos do programa*

- **list** ou **l** → lista o código fonte do arquivo, o list funciona como um cursor que mostra o código fonte da **linha A** até a **linha B**, quando ele é chamado novamente o list mostrara o código da **linha B+1** até a **linha C**
- **list n** ou **l n** → coloca o list para exibir código fonte do arquivo a partir de uma determinada linha (que será indicada por n), ou seja, o que este comando faz é mudar a **linha A** do comando list, o parâmetro **n** pode ser o nome de função também neste caso a **linha A** será o número da linha em que esta função começa.



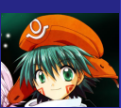
Relembrando o gdb

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Comandos gdb para visualizar o conteúdo de variáveis*

- **display variável** → faz com que o valor de uma variável seja exibido cada vez que o programa para sua execução.
- **info display** ou **i display** → mostra todos os displays que existem.
- **undisplay** → apaga todos os displays que existem.
- **undisplay idDisplay** → apaga o display com um determinado id.
- **print variável** ou **print variável** → mostra o valor de uma variável.



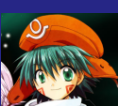
Relembrando o gdb

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Comandos relativos a pontos de parada*

- **break n** ou **b n** → coloca um ponto de parada na linha indicada por **n**, **n** pode ser também o nome de uma função neste caso o ponto de parada será colocado na primeira linha da função.
- **delete break** ou **d b** → deleta todos os pontos de parada existentes no programa.
- **info breakpoints** ou **i b** → lista todos os pontos de para existentes no programa.
- **delete idBreakPoint** ou **d idBreakPoint** → deleta um ponto de parada com determinado id, este id é especificado pelo parâmetro **idBreakPoint**.



Relembrando o gdb

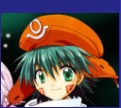
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Comandos relativos a pontos de parada*

- **condition idBreakPoint condição** → faz com que um determinado ponto de parada só se torne ativo caso uma determinada condição seja satisfeita, um exemplo de condição seria $x > 400$ (repare que variável x tem que existir no programa para esta condição ser declarada).

OBS: O * significa que o comando deve ser executado dentro do ambiente gdb e não no shell do SO.



CUDA-gdb

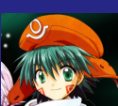
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

O CUDA gdb adiciona alguns possui alguns comandos a mais que o gdb, no entanto só é possível executar estes comando quando se esta dentro do código que roda na GPU.

Comandos extras do CUDA-gdb

- **CUDA block (X,Y) thread (X',Y',Z')** → este comando pode ser usado sempre que o programa esta parada dentro de um trecho da GPU ele permite que ir para uma determina copia da função que esta rodando dentro da GPU, nos exemplos mostrados nesta apresentação $0 \leq X < nblocks$, $0 \leq X' < blockSize$ e Y, Y' e Z' são iguais a 0.

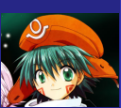


Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Comandos extras do CUDA-gdb

- **CUDA kernel idDaFunçãoCuda** → permite saber em que copia da função se esta no momento.
- **info CUDA system** → mostra informações sobre as placas de vídeo presentes no computador.
- **info CUDA kernels** → permite se saber as funções CUDA que estão atualmente rodando no computador.
- **info CUDA devices** → mostra todas as placas de vídeo do computador que estão executando alguma função CUDA e mostra quais funções CUDA elas estão executando.



CUDA-gdb

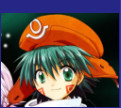
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Alguns comandos do cuda-gdb mudam um pouco sua funcionalidade quando o programa para dentro de um código que roda na GPU.

Comandos que mudam quando o CUDA-gdb para num código executado na GPU

- **next** ou **n** → faz com que a cópia da função em que se esta avance uma linha, as outras cópias da função permanecem na mesma linha.
- **Pontos de parada** → quando um ponto de parada é atingido todas as cópias da função param naquele ponto.
- **continue** ou **c** → quando executado faz com que todas as cópias da função prossigam sua execução até encontrarem um ponto de parada.



Compilando para o CUDA-gdb

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Para se compilar um código CUDA para que o mesmo possa ser aberto e debugado no CUDA-gdb utiliza-se um dos seguintes comandos:

Compilando em GPUs Fermi (geforce 4XX)

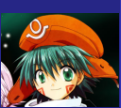
```
$ nvcc -g -G programaFonte.cu -gencode  
arch=compute_20,code=sm_20 -o programaexecutável
```

Compilando em GPUs Geforce 8,9,2XX

```
$ nvcc -g -G programaFonte.cu -o programaexecutável
```

Compilando em GPUs Tesla

```
$ nvcc -g -G programaFonte.cu -gencode  
arch=compute_10,code=sm_1 -gencode  
arch=compute_20,code=sm_20 -o programaexecutável
```



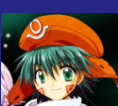
Abrir programa no CUDA-gdb

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Para se debugar o programa gerado pelos comandos anteriores utiliza-se o seguinte comando:

CUDA-gdb programaexecutável



Observações para se utilizar o CUDA-gdb

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

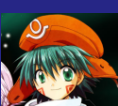
Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Importante *

Para executar o CUDA-gdb é necessário que exista duas placas de vídeo, uma placa cuidara da execução do servidor X e a outra placa cuidara da execução do código CUDA. Caso o servidor X esteja sendo executado pela placa de vídeo que rodara o código CUDA é necessário parar o servidor X para se poder utilizar o CUDA-gdb.

Importante **

Para se utilizar o CUDA-gdb é necessário que o servidor X tenha sido executado pelo menos uma vez enquanto o computador esta ligado. Isto é necessário porque para se rodar o código CUDA é preciso de alguns modulos do kernel do Linux, que só são executados quando o servidor X é ativado.



Front-ends gráficos para o CUDA-gdb

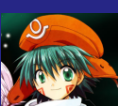
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Existem atualmente existem dois front-ends gráficos para o CUDA-gdb o `alinea ddt` e o `ddd`.

Alinea ddt

Um debugger pago que foi desenvolvido por uma empresa que possui uma longa tradição na criação de debuggers paralelos, é um front-end que incorpora nele todas as funções do CUDA-gdb, fazendo com que tudo na depuração seja wizard, algumas funções extras também foram adicionadas ao mesmo.



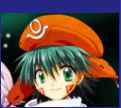
Front-ends gráficos para o CUDA-gdb

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

DDD

É um front-end opensource criado inicialmente para se usar junto com o gdb, sua interface em TK o faz parecer algo antigo, e sua integração com o CUDA-gdb não é completa pois todos os comandos extras existentes no CUDA-gdb não estão disponíveis através de sua interface, sendo preciso então executá-los via o shell do ddd.



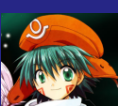
Utilizando o DDD localmente

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Caso o computador que ira rodar o código CUDA possua duas placas de vídeo, para se rodar o ddd nele execute o seguinte comando:

```
ddd -debugger CUDA-gdb programaexecutável
```



Utilizando o DDD remotamente

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

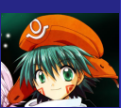
Rafael Guedes
Lang
Anderson
Gonçalves
Marco

Caso o computador que ira rodar o código CUDA possua apenas uma placa de vídeo é possível rodar o ddd remotamente, para isto crie um servidor ssh no computador que ira rodar o código CUDA e execute os seguintes comandos no computador cliente:

```
$ ssh -X -C ipDoComputadorQueIraRodarOcodigoCuda -l nomeDeUsuario
```

```
$ ddd -debugger CUDA-gdb programaexecutável
```

Um dos problema desta solução e que ela só funciona caso você tenha uma rede com um lag bem pequeno (normalmente só funciona bem em rede local), caso você esteja numa rede com um lag grande (como a internet) é possível executar o ddd via freenx (<http://www.nomachine.com/>).



Sumário

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco



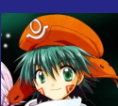
Matlab com CUDA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

o que é o Matlab

- É um software científico para computação numérica.



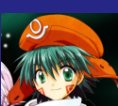
Matlab com CUDA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

o que é o Matlab

- É um software científico para computação numérica.
- Possui uma grande comunidade.



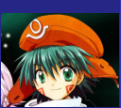
Matlab com CUDA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

o que é o Matlab

- É um software científico para computação numérica.
- Possui uma grande comunidade.
- Nele possível criar rotinas em C ou Fortran(estas rotinas são mais rápidas que as criadas na linguagem nativa do matlab).



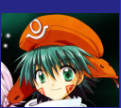
Matlab com CUDA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

o que é o Matlab

- É um software científico para computação numérica.
- Possui uma grande comunidade.
- Nele possível criar rotinas em C ou Fortran(estas rotinas são mais rápidas que as criadas na linguagem nativa do matlab).
- Possui um plug-in, disponibiliza-do pela nvidia, pra se fazer a ligação entre ele e o CUDA.



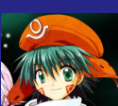
Matlab com CUDA

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

No próximo slide um exemplo de código em c, para matlab (sem usar CUDA), que pega o elementos de um vetor eleva cada elemento ao quadrado.

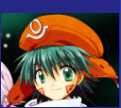
O nome do arquivo fonte é `square_matlab.c`



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

```
1  #include "mex.h"
2  void mexFunction(int nlhs, mxArray *plhs[],int nrhs, const mxArray *prhs[])
3  {
4      int i, j, m, n;
5      double *data1, *data2;
6      /* Checa se o numero de argumentos passados esta correto */
7      if (nrhs != 1 || nlhs !=1 )
8          mexErrMsgTxt("0 numero de argumentos de entrada ou de saida esta incorreto");
9
10     else{
11         /* Extrai as dimensoes da variavel de entrada*/
12         m = mxGetM(prhs[0]);
13         n = mxGetN(prhs[0]);
14         /* Cria uma variavel de saida */
15         plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
16         /* Cria-se um ponteiro para a variavel de entrada */
17         data1 = mxGetPr(prhs[0]);
18         /* Cria-se um ponteiro para variavel de saida */
19         data2 = mxGetPr(plhs[0]);
20         /* Faz a conta, e coloca o resultado na variavel de saida*/
21         for (j = 0; j < m*n; j++){
22             data2[j] = data1[j] * data1[j];
23         }
24     }
25 }
```

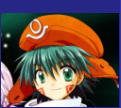


Compilando o arquivo e executando a rotina

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- 1 Para se compilar este arquivo entre no matlab, aponte o path do matlab para o diretorio onde esta o arquivo é digite `mex square_matlab.c`



Compilando o arquivo e executando a rotina

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

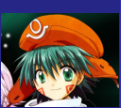
Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- 1 Para se compilar este arquivo entre no matlab, aponte o path do matlab para o diretorio onde esta o arquivo é digite `mex square_matlab.c`
- 2 Para se fazer a link do arquivo binário, resultante da compilação acima, com o matlab digite:

`which square_matlab square_matlab.mexw32` - Para Windows

`which square_matlab square_matlab.mexglx` - Para linux 32 bits

`which square_matlab square_matlab.mexa64` - Para linux 64 bits



Compilando o arquivo e executando a rotina

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

① Para se compilar este arquivo entre no matlab, aponte o path do matlab para o diretorio onde esta o arquivo é digite `mex square_matlab.c`

② Para se fazer a link do arquivo binário, resultante da compilação acima, com o matlab digite:

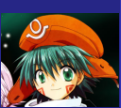
`which square_matlab square_matlab.mexw32` - Para Windows

`which square_matlab square_matlab.mexglx` - Para linux 32 bits

`which square_matlab square_matlab.mexa64` - Para linux 64 bits

③ Um exemplo de como executar a rotina no matlab:

`saida=square_matlab([1 2 3 4])`

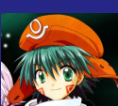


Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

No próximo slide um exemplo de código em CUDA, para matlab, que pega o elementos de um vetor eleva cada elemento ao quadrado.

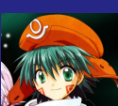
O nome do arquivo fonte é [square_matlab.cu](#)



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

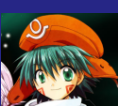
```
1  #include "cuda.h"
2  #include "mex.h"
3  /* Kernel to square elements of the array on the GPU */
4  __global__ void square_elements(float* in, float* out, int N){
5      int idx = blockIdx.x*blockDim.x+threadIdx.x;
6      if ( idx < N) out[idx]=in[idx]*in[idx];
7  }
8  /* Gateway function */
9  void mexFunction(int nlhs, mxArray *plhs[],int nrhs, const mxArray *prhs[]){
10     int j, m, n;
11     double *data1, *data2;
12     float *data1f, *data2f;
13     float *data1f_gpu, *data2f_gpu;
14     if (nrhs != 1 || nlhs != 1){
15         mexErrMsgTxt("0 numero de argumentos de entrada ou saida passados para a função");
16     }
17     else{
18         /* Find the dimensions of the data */
19         m = mxGetM(prhs[0]);
20         n = mxGetN(prhs[0]);
21         /* Create an mxArray for the output data */
22         plhs[0] = mxCreateDoubleMatrix(m, n, mxREAL);
23         /* Create an input and output data array on the GPU*/
24         cudaMalloc( (void **) &data1f_gpu, sizeof(float)*m*n);
25         cudaMalloc( (void **) &data2f_gpu, sizeof(float)*m*n);
26         /* Retrieve the input data */
27         data1 = mxGetPr(prhs[0]);
28         /* Check if the input array is single or double precision */
29         /* The input array is in double precision, it needs to be converted to floats before being processed */
30         data1f = (float *) mxMalloc(sizeof(float)*m*n);
31         for (j = 0; j < m*n; j++){
```



Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

```
32         data1f[j] = (float) data1[j];
33     }
34     cudaMemcpy( data1f_gpu, data1f, sizeof(float)*n*m, cudaMemcpyHostToDevice);
35     data2f = (float *) mxMalloc(sizeof(float)*m*n);
36     /* Compute execution configuration using 128 threads per block */
37     int dimBlock=128;
38     int dimGrid=((m*n)/dimBlock);
39     if ( (n*m) % 128 !=0 ) dimGrid+=1;
40     /* Call function on GPU */
41     square_elements<<<dimGrid,dimBlock>>>(data1f_gpu, data2f_gpu, n*m);
42     /* Copy result back to host */
43     cudaMemcpy( data2f, data2f_gpu, sizeof(float)*n*m, cudaMemcpyDeviceToHost);
44     /* Create a pointer to the output data */
45     data2 = mxGetPr(plhs[0]);
46     /* Convert from single to double before returning */
47     for (j = 0; j < m*n; j++){
48         data2[j] = (double) data2f[j];
49     }
50     /* Clean-up memory on device and host */
51     mxFree(data1f);
52     mxFree(data2f);
53     cudaFree(data1f_gpu);
54     cudaFree(data2f_gpu);
55 }
56 }
57
```



Compilando o arquivo e executando a rotina

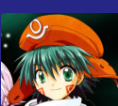
Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- 1 No shell do SO, dentro da pasta onde esta o arquivo fonte, digite o seguinte comando:

```
$diretorio_do_plug-in/nvmex -f $diretorio_do_plug-in/nvopts.sh square_matlab.cu -L
```

```
$diretorio_do_CUDA/lib -lcudart
```



Compilando o arquivo e executando a rotina

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- 1 No shell do SO, dentro da pasta onde esta o arquivo fonte, digite o seguinte comando:

```
$diretorio_do_plug-in/nvmex -f $diretorio_do_plug-in/nvopts.sh square_matlab.cu -L
```

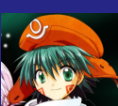
```
$diretorio_do_CUDA/lib -lcudart
```

- 2 Para se fazer a link do arquivo binário, resultante da compilação acima, entre no matlab, aponte o path do matlab para o diretorio onde esta o arquivo é digite:

`which square_matlab square_matlab.mexw32` - Para Windows

`which square_matlab square_matlab.mexglx` - Para linux 32 bits

`which square_matlab square_matlab.mexa64` - Para linux 64 bits



Compilando o arquivo e executando a rotina

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

- 1 No shell do SO, dentro da pasta onde esta o arquivo fonte, digite o seguinte comando:

```
$diretorio_do_plug-in/nvmex -f $diretorio_do_plug-in/nvopts.sh square_matlab.cu -L
```

```
$diretorio_do_CUDA/lib -lcudart
```

- 2 Para se fazer a link do arquivo binário, resultante da compilação acima, entre no matlab, aponte o path do matlab para o diretorio onde esta o arquivo é digite:

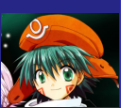
`which square_matlab square_matlab.mexw32` - Para Windows

`which square_matlab square_matlab.mexglx` - Para linux 32 bits

`which square_matlab square_matlab.mexa64` - Para linux 64 bits

- 3 Um exemplo de como executar a rotina no matlab:

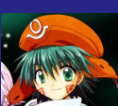
```
saida=square_matlab([1 2 3 4])
```



Sumário

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco



Bibliografia

Utilização da
unidade
processamento
gráfico para
propósito
geral
(GPGPU)

Rafael Guedes
Lang
Anderson
Gonçalves
Marco

www.nvidia.com/CUDA

<http://forums.nvidia.com/index.php?showforum=62>

www.gpGPU.org

www.clubedohardware.com.br

www.mathematik.uni-dortmund.de/~goeddeke/arcs2008/C1_CUDA.pdf

home.in.tum.de/~heinecke/docs/heinecke_fa2007_slides.pdf

NVIDIA CUDA Programming Guide