Introdução à Computação Paralela em Sistemas Heterogêneos

Aula 019 - Diretivas de Compilação para *Offloading* de Código para Dispositivos Aceleradores

Prof. Rogério Aparecido Gonçalves[1]
rogerioag@utfpr.edu.br

25 de abril de 2021

[1] Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento de Computação (DACOM)
Campo Mourão - Paraná - Brasil

Ciência da Computação
PD36O - Paradigmas de Programação Paralela e Distribuída

Agenda i

1. Introdução
2. Suporte à Offloading de Código para Aceleradores
3. *Kernels* para GPU com CUDA
4. Padrão OpenACC
5. Suporte a *Offloading* de Código para Aceleradores no OpenMP
6. Atividades sobre *target*
7. Dúvidas

Agenda ii

8. Referências

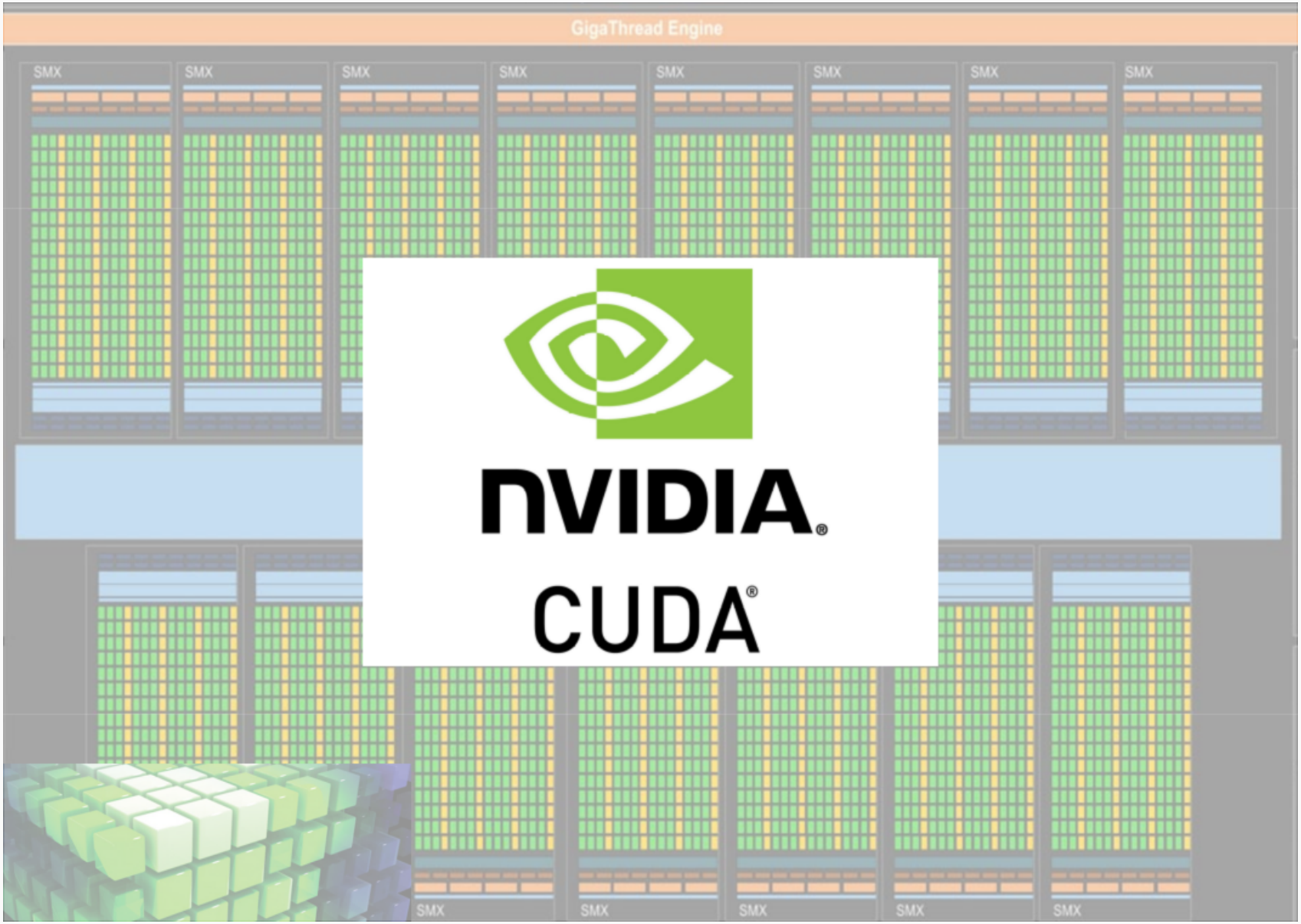
Introdução

Objetivos

- Explorar os Construtores para *offloading* de código para dispositivos aceleradores. O padrão *OpenACC* e *OpenMP* **target**.
- Testar os efeitos do construtor `target` e as cláusulas admitidas[1], para *offloading* de código.

[1](OpenMP-ARB 2013)(OpenMP-ARB 2014)

Suporte à Offloading de Código para Aceleradores


GigaThread Engine
SMX
NVIDIA
CUDA

Kernels para GPU com **CUDA**

Função `kernel` para GPU i

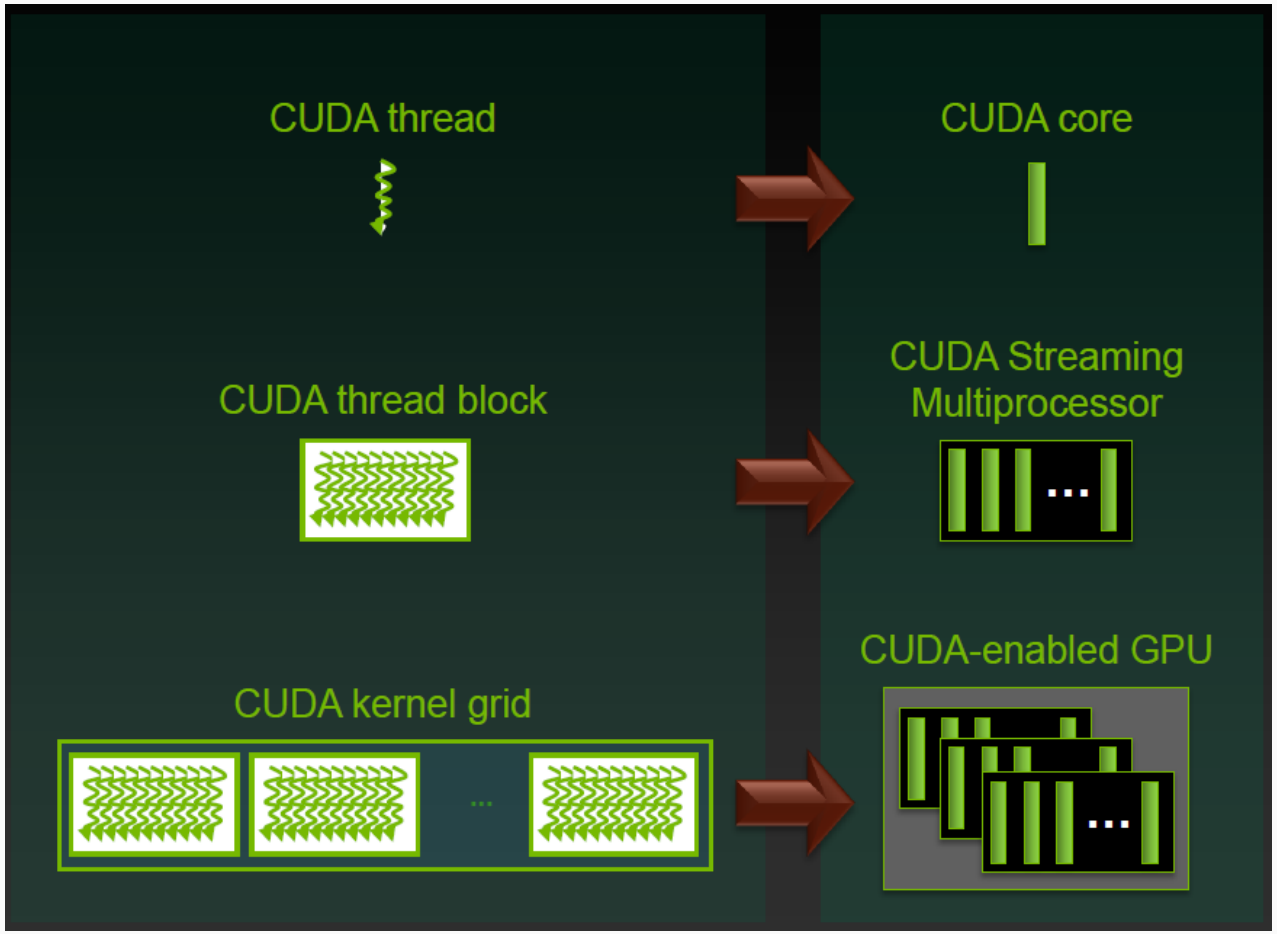


Figure 1: Execução de uma função `kernel`

Função `kernel` para GPU ii

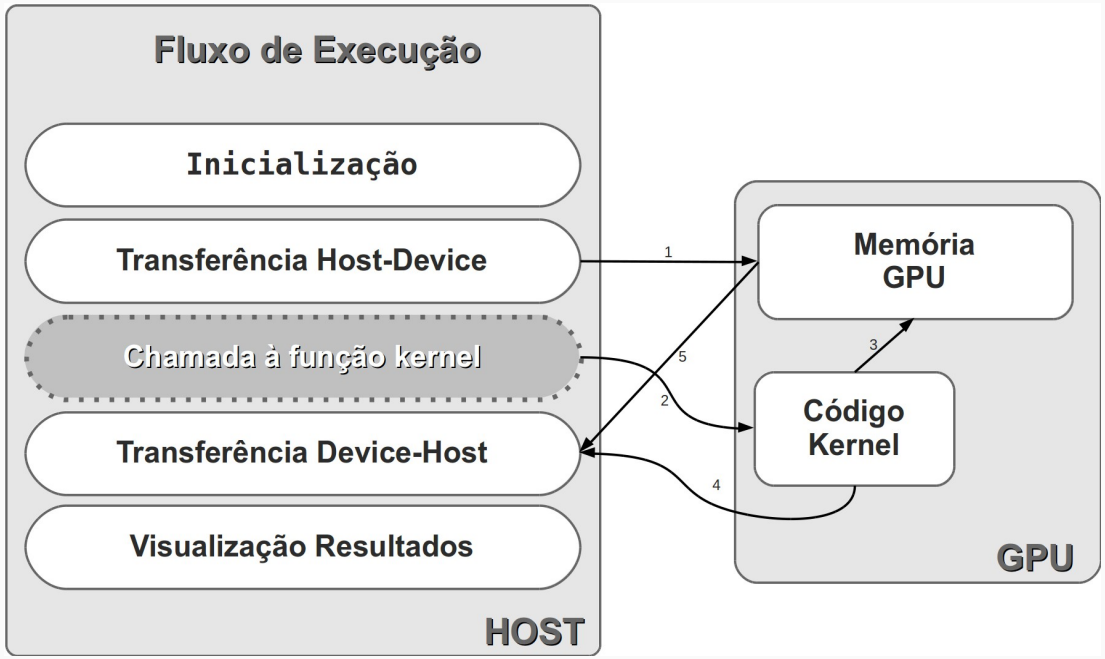


Figure 2: Modelo Clássico

Função `kernel` para GPU iii

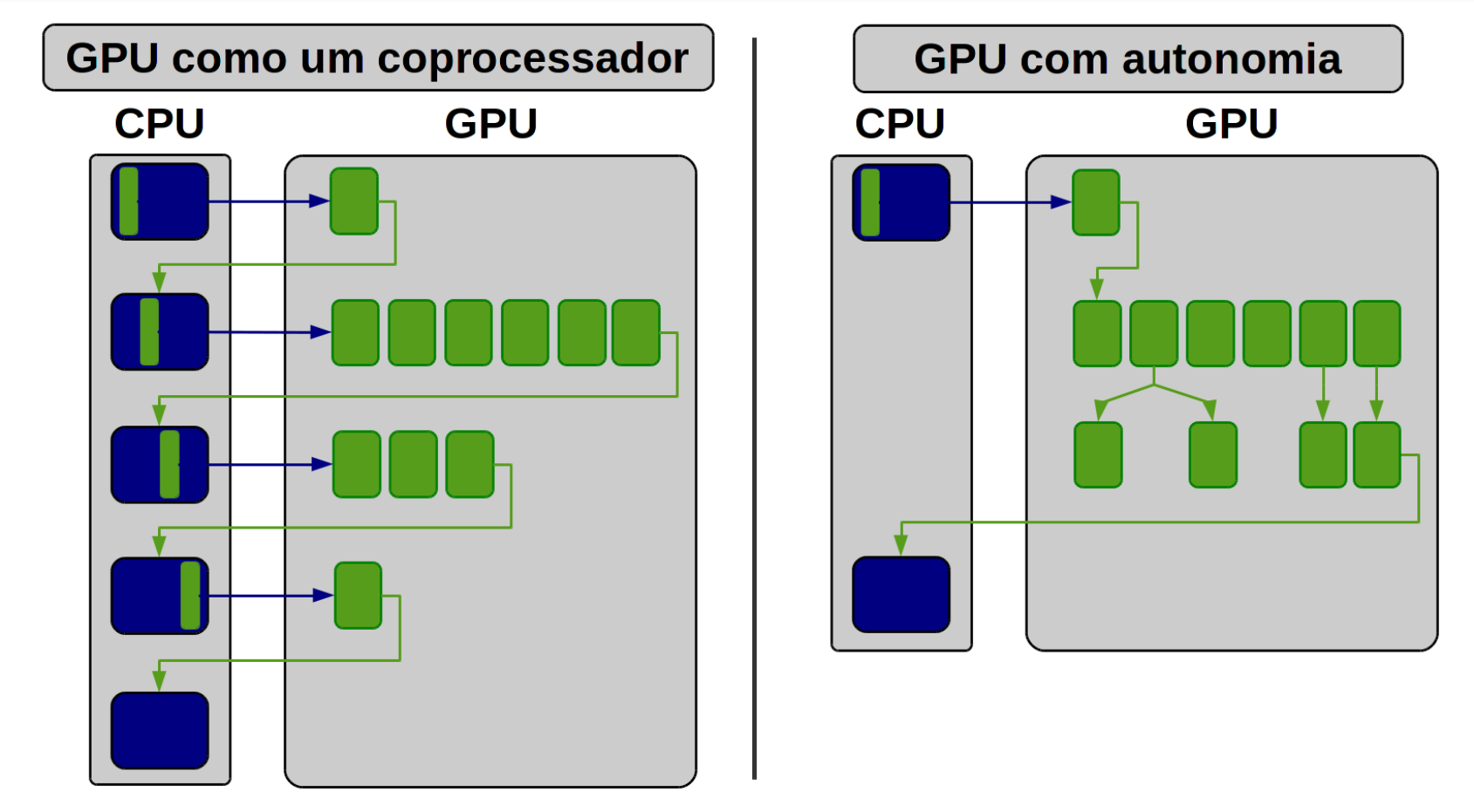


Figure 3: Paralelismo Diâmico

Função kernel para GPU i

- Para falarmos sobre **diretivas de compilação** para aceleradores temos que introduzir o modelo de programação para aceleradores como as **GPUs**.
- Para esse tipo de dispositivo acelerador é necessário definir uma função *kernel* que terá sua execução lançada no dispositivo.

```
__global__ void vecAdd(float *a, float *b, float *c, int n)
{
  int id = blockIdx.x * blockDim.x + threadIdx.x;
  if (id < n)
    c[id] = a[id] + b[id];
}
```

Função kernel para GPU ii

- Declaração dos dados e ponteiros no dispositivo:

```
int main( int argc, char* argv[] ){
  float *h_a;
  float *h_b;
  float *h_c;

  // Declaração dos vetores de entrada na memória da GPU.
  float *d_a;
  float *d_b;
  // Declaração do vetor de saída do dispositivo.
  float *d_c;

  // Tamanho em bytes de cada vetor.
  size_t bytes = n * sizeof(float);
```

Função kernel para GPU iii

```
// Alocação de memória para os vetores do host.
h_a = (float*) malloc(bytes);
h_b = (float*) malloc(bytes);
h_c = (float*) malloc(bytes);

// Alocação de memória para cada vetor na GPU.
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);
```

Função `kernel` para GPU iv

- **CUDA** fornece função para realizar transferências de dados entre a memória principal e a memória do dispositivo.

```
// Cópia dos vetores do host para o dispositivo.
cudaMemcpy( d_a, h_a, bytes, cudaMemcpyHostToDevice);
cudaMemcpy( d_b, h_b, bytes, cudaMemcpyHostToDevice);
```

- Então é feita a chamada à função **kernel**.
- Na ativação do *kernel* a configuração da estrutura do arranjo de *threads* `(grid e bloco)` precisa ser definida explicitamente pelo programador.
- Essa configuração determina quantas *threads* serão criadas e como estarão organizadas em blocos dentro do *grid* mapeado para o dispositivo.

Função kernel para GPU v

```
int blockSize, gridSize;

// Número de threads em cada bloco de threads.
blockSize = 1024;

// Número de blocos de threads no grid.
gridSize = (int)ceil((float)n/blockSize);

// Chamada a função kernel.
vecAdd<<<gridSize, blockSize>>>(d_a, d_b, d_c, n);
```

Função `kernel` para GPU vi

- Cópia do resultado `(d_c)` da soma de vetores realizada no dispositivo para `(h_c)` na memória do *host*.
- Liberação da memória alocada.

```
// Cópia do vetor resultado da GPU para o host.
cudaMemcpy(h_c, d_c, bytes, cudaMemcpyDeviceToHost );

// Liberação da memória da GPU.
cudaFree(d_a); cudaFree(d_b); cudaFree(d_c);

// Liberação da Memória do host.
free(h_a); free(h_b); free(h_c);

return 0;
```

Padrão OpenACC


More Science, Less Programming
About Tools News Stories Events Resources Spec Community
What is OpenACC?
OpenACC is a user-driven directive-based performance-portable parallel programming model designed for scientists and engineers interested in porting their codes to a wide-variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required with a low-level model.
or take the next steps
#pragma acc data copy(A) create(Anew)
while (error > tol && iter < iter_max) {
error = 0.0;
#pragma acc kernels {
#pragma acc loop independent collapse(2)
for (int j = 1; j < n-1; j++) {
for (int i = 1; i < m-1; i++) {
Anew [j] [i] = 0.25 * (A [j] [i+1] + A [j] [i-1] +
A [j-1] [i] + A [j+1] [i])
;
error = max (error, fabs (Anew [j] [i] - A [j] [i]));
}
}
}
OpenACC
Directives for Accelerators
Resources
Access tutorials, guides, lectures, code samples, hands-on exercises and more.
Get the Specs
Tools
Success Stories
Latest News
OpenACC October Highlights
Nov. 1, 2017
PGI 17.7 Is Now Available
Aug. 14, 2017
Press Release: OpenACC Flying High at ISC17
Jun. 19, 2017
Upcoming Events
SC17
November 12, 2017
Denver, Colorado, USA
Scalable Parallel Programming Using OpenACC for Multicore, GPUs, and Manycore
November 13, 2017
SC17, Denver, Colorado
Join Us
Tweets by @OpenACCorg
OpenACC @OpenACCorg
Congrats to WACCPD best paper award winners! This year 2 #OpenACC papers received an award! #SC17
OpenAC

Offloading para Aceleradores: construtor `target` i

- No contexto de *diretivas de compilação* o padrão **OpenACC** fornece um conjunto de diretivas.
- Anunciado em novembro de 2011 na conferência SuperComputing.
- É um padrão para programação paralela.
- A ideia é anotar o código como o **OpenMP** e *kernels* para **GPU** serem gerados.
- Recursos: https://www.openacc.org/resources
- Especificação: https://www.openacc.org/sites/default/files/inline-files/OpenACC_2pt5_0.pdf

Offloading para Aceleradores: construtor `target` ii

- Descreve uma API de programação que fornece uma coleção de diretivas para especificar laços e regiões de código paralelizáveis que podem ter sua execução acelerada por dispositivo tal como uma GPU.
- As Diretivas em C/C++ são especificadas usando `#pragma` como no OpenMP. Se o compilador não utilizar pré-processamento, as anotações são ignoradas na compilação.
- Cada diretiva em C/C++ inicia com `#pragma acc` e existem construtores e cláusulas para a criação de *kernels* com base em laços.
- Laços paralelizáveis anotados para serem transformados em *kernel* para um dispositivo.

Diretivas do OpenACC

```
#pragma acc directive-name [clause [[,] clause]...] new-line
```

Offloading para Aceleradores: construtor `target` iii

- O exemplo soma de vetores escrito com as diretivas do OpenACC.

```
void vecaddgpu(float *restrict c, float *a, float *b, int n){
  #pragma acc kernels for present(c,a,b)
  for( int i = 0; i < n; ++i )
    c[i] = a[i] + b[i];
}

int main( int argc, char* argv[] ){

  #pragma acc data copyin(a[0:n],b[0:n]) copyout(c[0:n])
  {
    vecaddgpu(c, a, b, n);
  }

  return 0;
}
```

- A saída gerada pelo compilador `pgcc`:

Offloading para Aceleradores: construtor `target` iv

Terminal

```
rag@chamonix:/src/example-openacc$ pgcc -acc -ta=nvidiatime -Minfo=accel
     -fast vectoradd.c -o vectoradd-acc-gpu
vecaddgpu:
12 Generating present(b[0:])
Generating present(a[0:])
Generating present(c[0:])
Generating compute capability 1.0 binary
Generating compute capability 2.0 binary
13 Loop is parallelizable
Accelerator kernel generated
13 #pragma acc loop gang vector(256) /* blockIdx.x threadIdx.x */
CC 1.0 : 5 registers; 36 shared 4 constant 0 local memory bytes; 100%
     occupancy
CC 2.0 : 5 registers; 4 shared 48 constant 0 local memory bytes; 100%
     occupancy
main:
44 Generating copyout(c[0:n])
Generating copyin(b[0:n])
Generating copyin(a[0:n])
rag@chamonix:/src/example-openacc$
```

Suporte a *Offloading* de Código para Aceleradores no OpenMP

Construtor `target` i

- A Diretiva: `#pragma omp target`
- Para *offloading* de código para dispositivos aceleradores, no `OpenMP` temos o construtor target.

Sintaxe

```
#pragma omp target [clause[ [ , ] clause] ... ] new-line
bloco-estruturado
```

- O Código apresenta os construtores `target` e `parallel for` combinados.
- O construtor `target` faz o mapeamento de variáveis para a memória do dispositivo e lança a execução do código no dispositivo.
- Uma função com o código associado ao construtor `target` é criada para ser executada no dispositivo alvo.

Construtor `target` ii

- O dispositivo alvo *(device target)* pode ser definido chamando a função `omp_set_default_device(int device_num)` com o número do dispositivo sendo passado como argumento ou definindo-se a variável de ambiente `OMP_DEFAULT_DEVICE` ou ainda usando a cláusula `device(device_num)`.

Construtor `target` iii

- **Exemplo:** Soma de vetores.

```
void vecaddgpu(float *c, float *a, float *b, int n){
  #pragma omp target device(0)
  #pragma omp parallel for private(i)
  for( int i = 0; i < n; ++i ){
    c[i] = a[i] + b[i];
  }
}
```

Construtor `target` iv

- O mapeamento de dados para o dispositivo pode ser feito usando-se a cláusula `map` admitida pelo construtor `target`.

```
void vecaddgpu(float *c, float *a, float *b, int n){
  #pragma omp target map(to: a[0:n], b[:n]) map(from: c[0:n])
  #pragma omp parallel for private(i)
  for( int i = 0; i < n; ++i ) {
    c[i] = a[i] + b[i];
  }
}
```

Construtor `target v`

- O construtor **`target`** também permite a escolha de fazer o *offloading* do código para o dispositivo ou não, utilizando a cláusula **`if`**.
- As transferências de dados também podem ser declaradas com o construtor **`target data`** que cria um novo ambiente de dados que será utilizado pelo *kernel*.

```
#define THRESHOLD 1024

void vecaddgpu(float *c, float *a, float *b, int n){
  #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:n]) if(n>THRESHOLD)
  {
    #pragma omp target if(n>THRESHOLD)
    #pragma omp parallel for if(n>THRESHOLD)
    for( int i = 0; i < n; ++i )
      c[i] = a[i] + b[i];
  }
}
```

Construtor `target` vi

- Especificar uma região de dados pode ser útil para múltiplos *kernels*

```
#define THRESHOLD 1048576

void vecaddgpu(float *c, float *a, float *b, int n){
  #pragma omp target data map(from: c[0:n])
  {
    #pragma omp target if(n>THRESHOLD) map(to: a[0:n], b[:n])
    #pragma omp parallel for
    for( int i = 0; i < n; ++i )
      c[i] = a[i] + b[i];

    // Reinicialização dos dados.
    init(a,b);
    #pragma omp target if(n>THRESHOLD) map(to: a[0:n], b[:n])
    #pragma omp parallel for
    for( int i = 0; i < n; ++i )
      c[i] = c[i] + (a[i] * b[i]);
  }
}
```

Construtor `target` vii

- Atualização dos dados entre as execuções dos *kernels* é utilizando o construtor `target update`.

```
void vecaddgpu(float *c, float *a, float *b, int n){
  int changed = 0;
  #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:n])
  {
    #pragma omp target
    #pragma omp parallel for
    for( int i = 0; i < n; ++i )
      c[i] = a[i] + b[i];

    changed = init(a,b);
    #pragma omp target update if(changed) to(a[0:n], b[:n])
    #pragma omp target
    #pragma omp parallel for
    for( int i = 0; i < n; ++i )
      c[i] = c[i] + (a[i] * b[i]);
  }
}
```

Construtor `target` viii

- O exemplo de soma de vetores feito **OpenMP** utilizando o construtor `target` e suas combinações vistas nos exemplos anteriores.

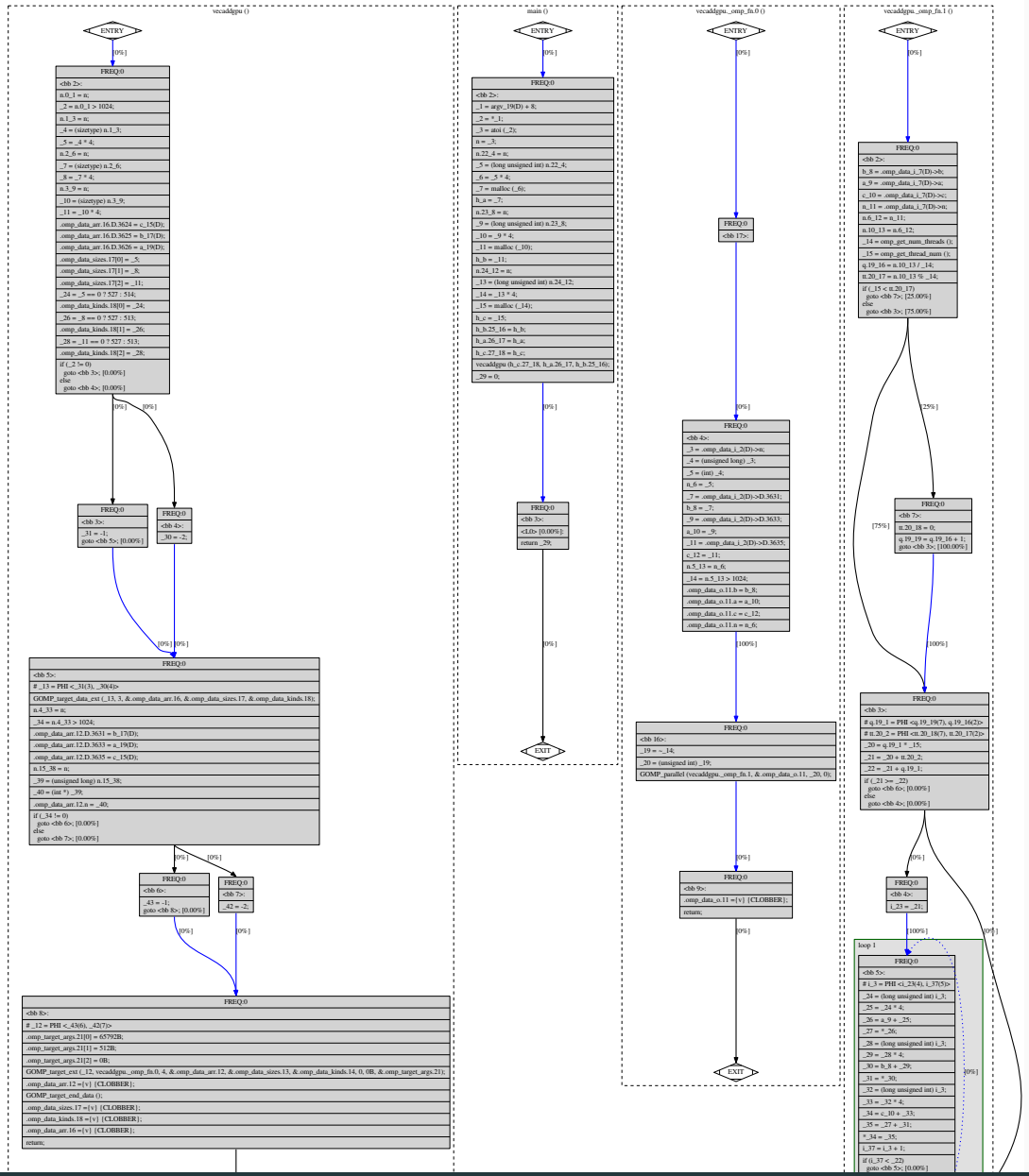
```
#define THRESHOLD 1024

float *h_a; float *h_b; float *h_c;
int n = 0;
/* Código Suprimido. */
void vecaddgpu(float *c, float *a, float *b){
  #pragma omp target data map(to: a[0:n], b[:n]) map(from: c[0:n]) if(n>THRESHOLD)
  {
    #pragma omp target if(n>THRESHOLD)
    #pragma omp parallel for if(n>THRESHOLD)
    for( int i = 0; i < n; ++i ){
      c[i] = a[i] + b[i];
    }
  }
}
```

Construtor `target` ix

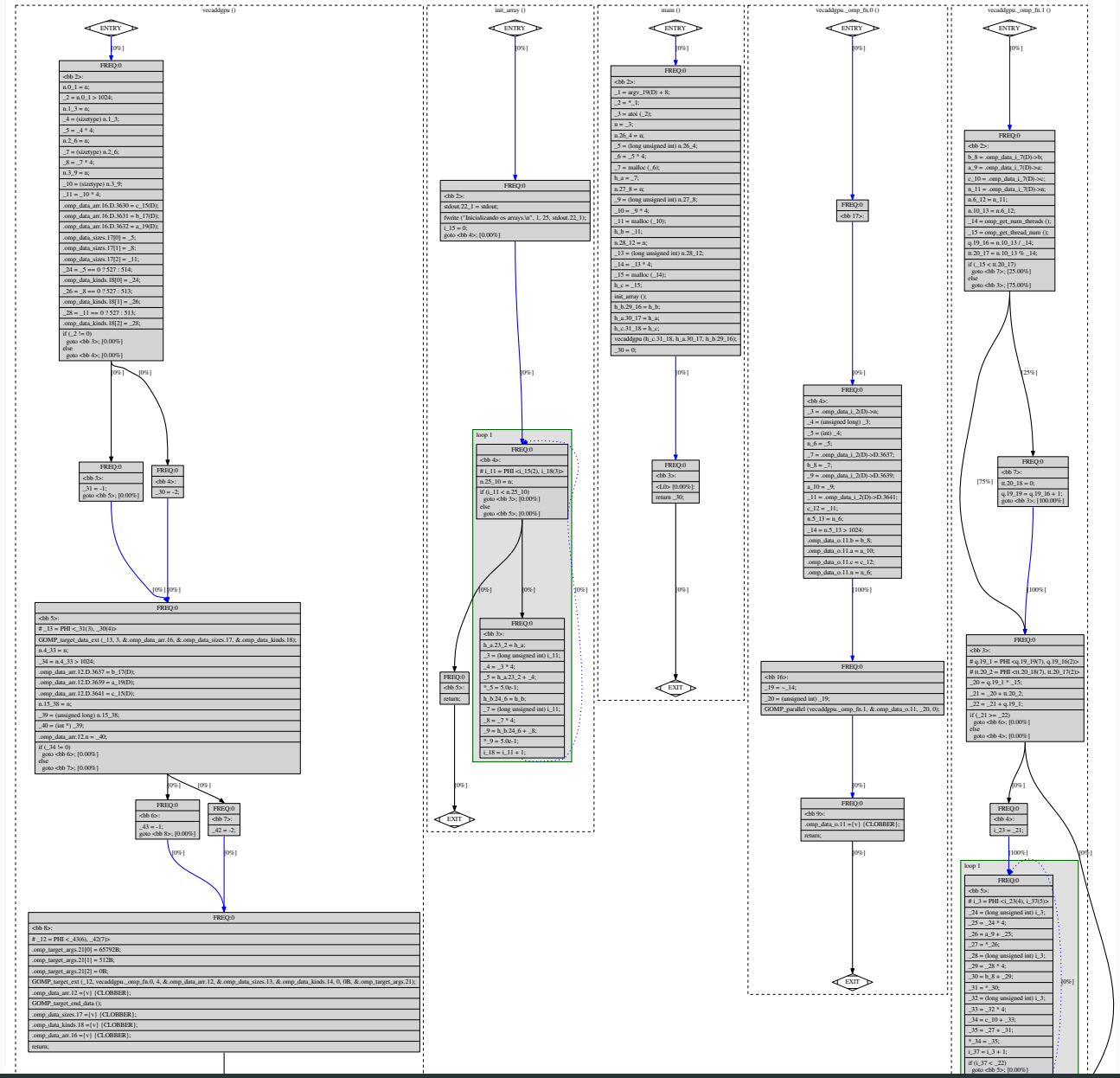
- O exemplo de soma de vetores feito OpenMP utilizando o construtor `target` e suas combinações vistas nos exemplos anteriores.

```
int main(int argc, char *argv[]) {
  int i;
  n = atoi(argv[1]);

  h_a = (float*) malloc(n*sizeof(float));
  h_b = (float*) malloc(n*sizeof(float));
  h_c = (float*) malloc(n*sizeof(float));

  init_array();

  vecaddgpu(h_c, h_a, h_b);

  return 0;
}
```

Estrutura do Código gerado para o construtor `target`



Estrutura do Código gerado para o construtor `target`



Suporte da ABI da LIBGOMP i

- As funções relacionadas com a geração de código para o construtor `target` que identificamos na ABI da `libgomp`:

ABI libgomp – Funções relacionadas com o construtor target

```
void GOMP_parallel (void (*fn) (void *), void *data, unsigned num_threads,
unsigned int flags)
void GOMP_target_data_ext (int device, size_t mapnum, void **hostaddrs,
size_t *sizes, unsigned short *kinds)
void GOMP_target_end_data (void)
void GOMP_target_update (int device, const void *unused, size_t mapnum,
void **hostaddrs, size_t *sizes, unsigned char *kinds)
void GOMP_target_ext (int device, void (*fn) (void *), size_t mapnum,
void **hostaddrs, size_t *sizes, unsigned short *kinds, unsigned int flags,
void **depend, void **args)
```

Exemplo de Execução i

- Como o código utiliza a cláusula `if` para decidir se deve ou não fazer o *offloading* com base no tamanho dos dados.

Exemplo de Execução ii

Terminal

```
rag@ragserver:src/example-target$ nvprof ./example-target.exe 16384
Inicializando os arrays.
==2381== NVPROF is profiling process 2381 command: ./example-target.exe
    16384
Verificando o resultado.
Resultado Final: (16384.000000 1.000000)
==2381== Profiling application: ./example-target.exe 16384
==2381== Profiling result:
.
Time(%) Time Calls Avg Min Max Name
97.56% 2.6697ms 1 2.6697ms 2.6697ms 2.6697ms vecaddgpu$_omp_fn0 1.61%
    44.160us 6 7.3600us 1.0560us 21.408us [CUDA memcpy HtoD] 0.82%
    22.496us 1 22.496us 22.496us 22.496us [CUDA memcpy DtoH] ==2381== API
    calls: Time(%) Time Calls Avg Min Max Name 60.98% 131.59ms 1 131.59ms
    131.59ms 131.59ms cuCtxCreate 34.12% 73.631ms 1 73.631ms 73.631ms
    73.631ms cuCtxDestroy 1.24% 2.6735ms 1 2.6735ms 2.6735ms 2.6735ms
    cuCtxSynchronize 1.17% 2.5168ms 22 114.40us 32.989us 999.11us
    cuLinkAddData 1.07% 2.3177ms 1 2.3177ms 2.3177ms 2.3177ms
    cuModuleLoadData 0.45% 961.91us 1 961.91us 961.91us 961.91us
    cuLinkComplete 0.25% 544.36us 1 544.36us 544.36us 544.36us
    cuLaunchKernel 0.18% 388.84us 3 129.61us 125.77us 135.86us cuMemAlloc
```

Exemplo de Execução i

- Da mesma forma o exemplo foi executado com $n = 512$ e podemos verificar com o `nvprof` que o *offloading* de código não foi feito.

Terminal

```
rag@ragserver:~/example-target$ nvprof ./example-target.exe 512
Inicializando os arrays.
Verificando o resultado.
Resultado Final: (512.000000 1.000000)
======== Warning: No CUDA application was profiled exiting
```

- Nenhuma operação relacionada ao dispositivo (transferências de dados e lançamento da execução de *kernels*) que caracterizaria o *offloading* de código aconteceu.

Atividades sobre *target*

Exercício de Implementação

1. Executar os exemplos para testar o *offloading* de código para GPU utilizando o OpenMP.

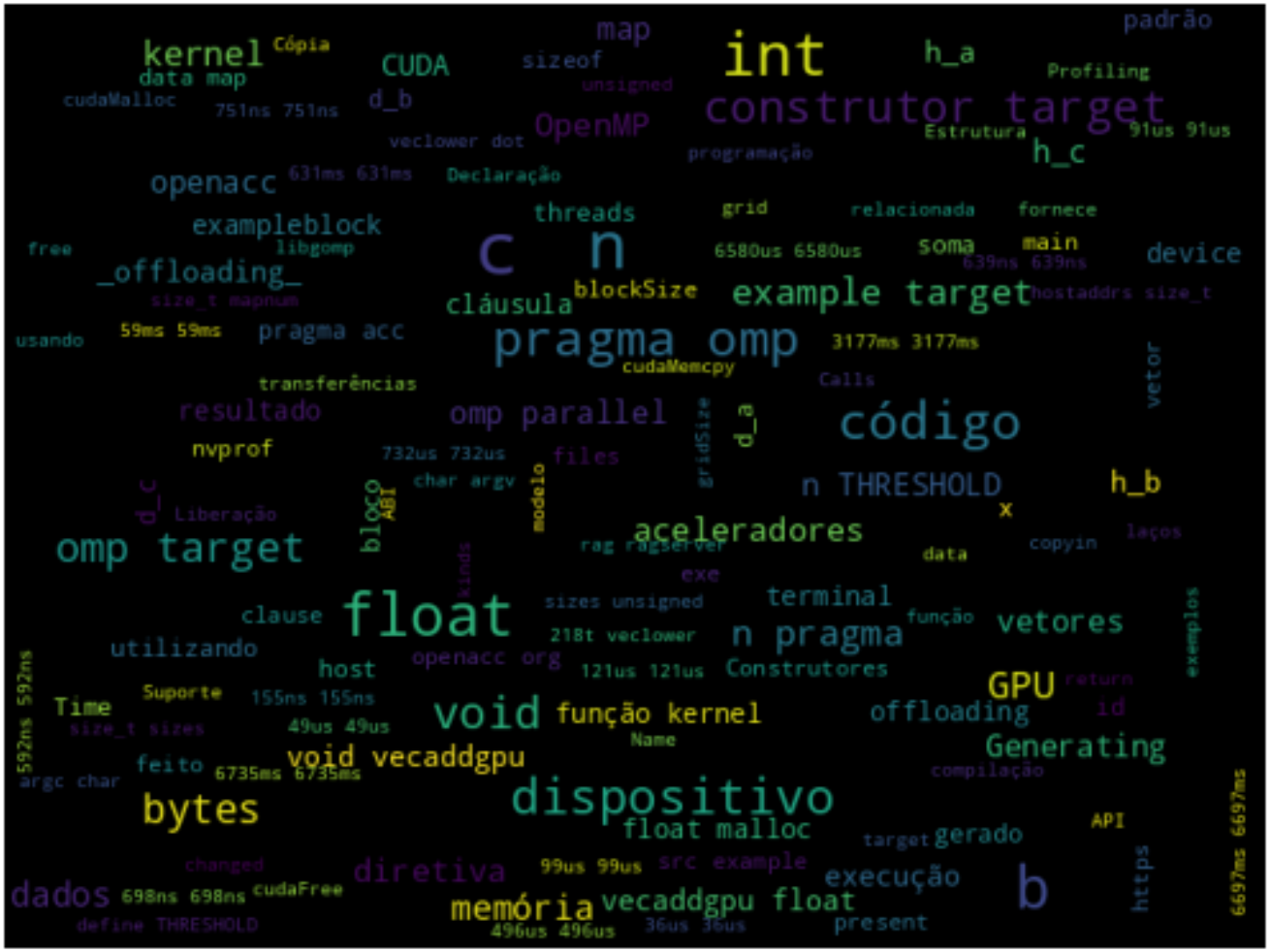
Dúvidas

Perguntas?

Prof. Rogério Aparecido Gonçalves

rogerioag@utfpr.edu.br

Word Cloud


kernel
Cópia
data map
cudaMalloc
751ns 751ns
CUDA
d_b
map
sizeof
unsigned
int
h_a
padrão
Profiling
construtor target
Estrutura
91us 91us
h_c
OpenMP
veclower dot
programação
openacc
631ms 631ms
Declaração
exampleblock
libgomp
free
threads
grid
relacionada
fornece
c n
6580us 6580us
soma
main
639ns 639ns
device
offloading
size_t mapnum
cláusula
blockSize
example target
hostaddrs size_t
usando
59ms 59ms
pragma acc
pragma omp
3177ms 3177ms
cudaMemcpy
Calls
vetor
transferências
resultado
omp parallel
d_a
código
nvprof
732us 732us
files
gridSize
char argv
n THRESHOLD
h_b
x
Liberação
bloco
ABI
modelo
aceleradores
omp target
rag ragserver
copyin
laços
data
exe
float
sizes unsigned
terminal
clause
função
vetores
218t veclower
n pragma
utilizando
openacc org
121us 121us
Construtores
host
GPU
Suporte
155ns 155ns
592ns 592ns
Time
void
função kernel
return
id
offloading
size_t sizes
49us 49us
Name
Generating
void vecaddgpu
feito
6735ms 6735ms
argc char
compilação
bytes
dispositivo
float malloc
API
target
gerado
changed
diretiva
99us 99us
src example
execução
b
https
6697ms 6697ms
dados
698ns 698ns
cudaFree
memória
vecaddgpu float
496us 496us
36us 36us
present
define THRESHOLD

Referências

Referências i

OpenMP-ARB. 2013. *OpenMP Application Program Interface Version 4.0*. OpenMP Architecture Review Board (ARB). http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf.

———. 2014. *OpenMP Application Program Interface Examples 4.0.1*. OpenMP Architecture Review Board (ARB). http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf.