



Escola Supercomputador SDUMONT

MC07-SD: Introdução à Programação em Aceleradores com Diretivas

Pedro Pais Lopes

12/02/2019

Agenda

- Parte 1: teoria
- Parte 2: compiladores
- Parte 3: pré-prática
- Parte 4: prática

Parte 1: TEORIA

- O acelerador
- Por que acelerar programas por diretivas?
- OpenACC e OpenMP target
- Motivação e portabilidades
- Modelos de memória e execução
- Diretivas
- Pequeno exemplo

Ponto de largada!

Aceleração de programas não
é pra você

N M M Meu programa não tem laço!





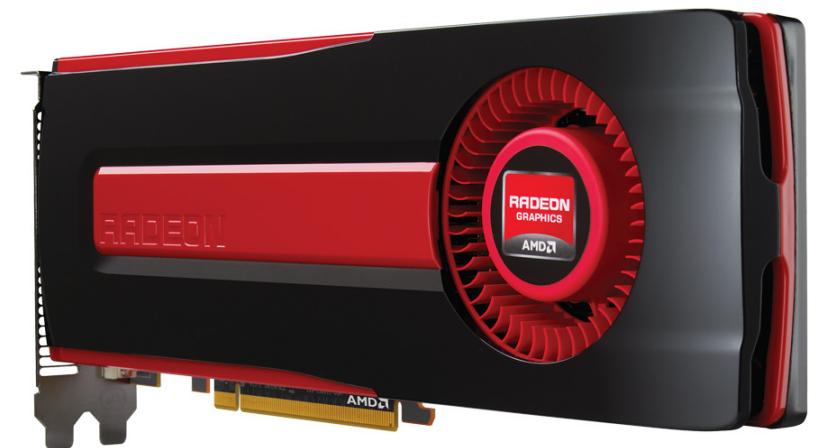
O acelerador

- Acelerador: dispositivo que realiza cálculos matemáticos massivos, muito específico e externo ao conjunto CPU-Memória
- Em geral a inequação abaixo é respeitada

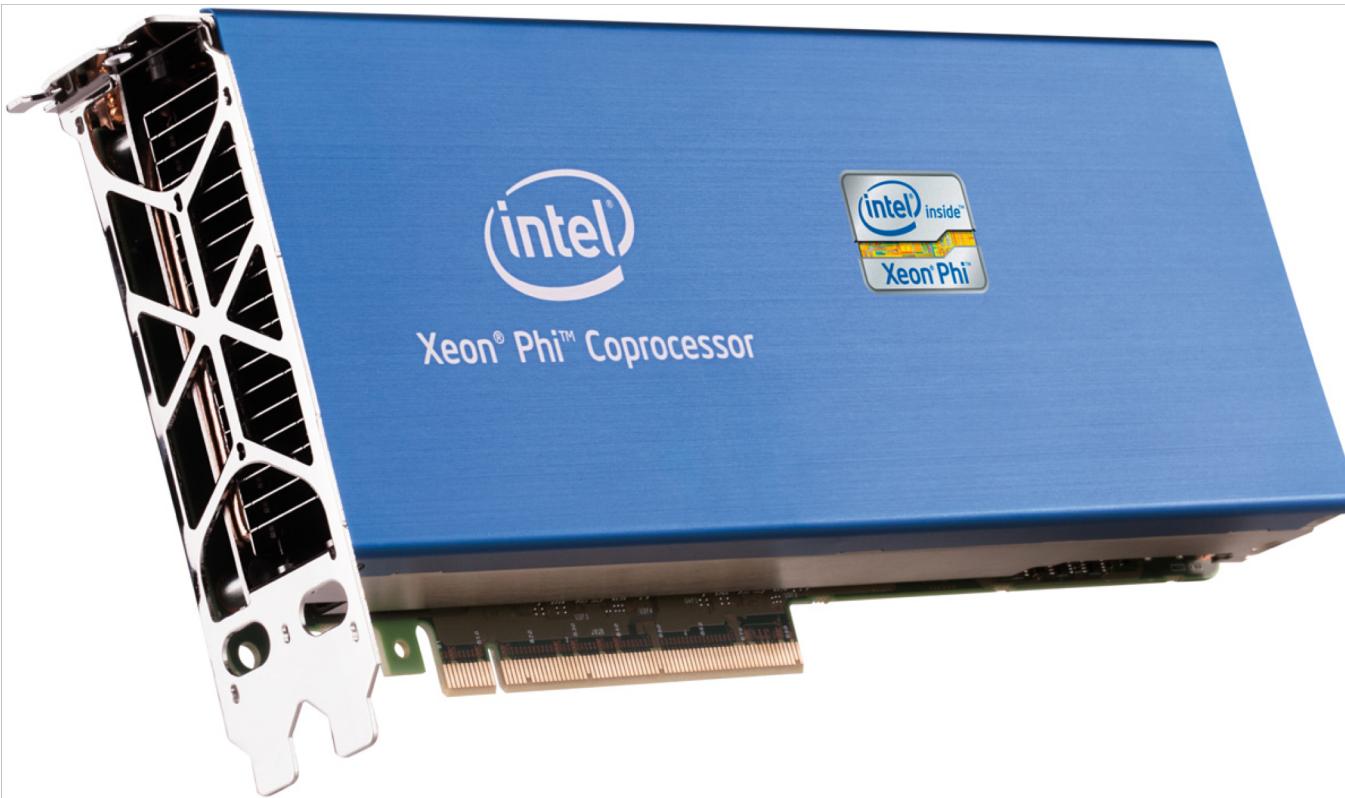
$$n\text{Cores}_{\text{acelerador}} \ggg n\text{Cores}_{\text{CPU}}$$

Exemplo

- Mais famoso: GPGPU (General Purpose computation on Graphics Processing Units)



Outro exemplo





40.960 CORES = 168.000.000.000 transistores (GPU)

44 CORES = 14.400.000.000 transistores (CPU)

399 mil dólares / 2 teraFLOPS
(pergunte-me como!)

RS\$ 1.600.000 (FOB)

ou

RS\$ 2.800.000 (TAX)

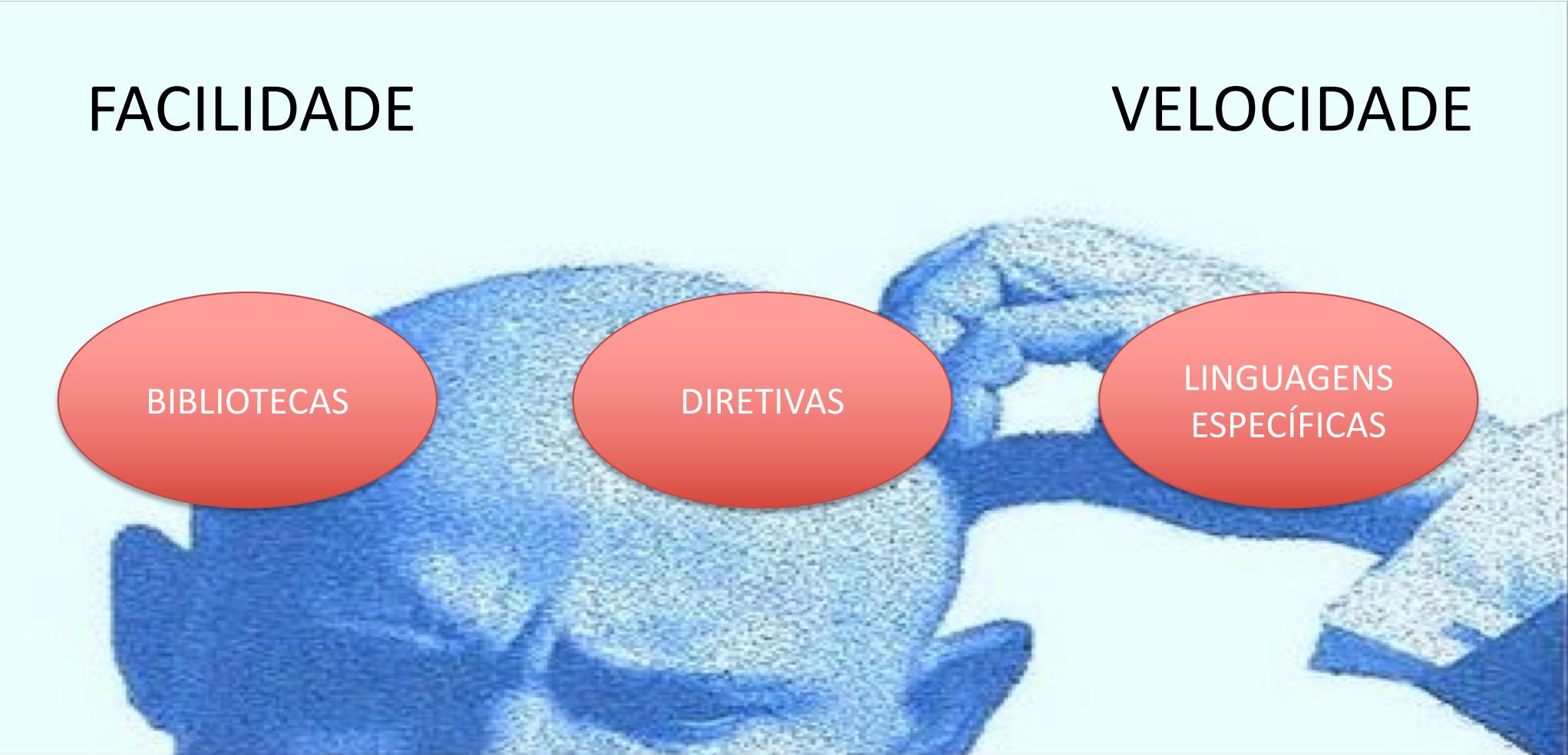
Assim!

	DGX-1P	DGX-1V	%	DGX-2	%
Preço	129.000	149.000	15,5%	399.000	167,8%
Int8 Teraops		480		1.005	109,3%
\$/Teraops		310		397	27,9%
FP16Teraflops	170	960	466,0%	2.000	108,3%
\$/FP16Teraflops	761	155	-79,6%	200	28,5%
FP32Teraflops	85	120	41,5%	251	109,3%
\$/FP32Teraflops	1.521	1.242	-18,4%	1.588	27,9%
FP16/FP32	2	8		8	
FP64Teraflops	43	60	41,2%	125	108,0%
\$/FP64Teraflops	3.035	2.483	-18,2%	3.197	28,7%
FP32/FP64	2	2		2	

Quero acelerar meu código, o que escolho?

FACILIDADE

VELOCIDADE



BIBLIOTECAS

DIRETIVAS

LINGUAGENS
ESPECÍFICAS

Como levar este poder de processamento ABUNDANTE para sua aplicação?

DIFÍCULDADE
INTRUSÃO AO CÓDIGO
DESEMPENHO



Matrix Multiply Source Code

Size Comparison:

Directives

CUDA C

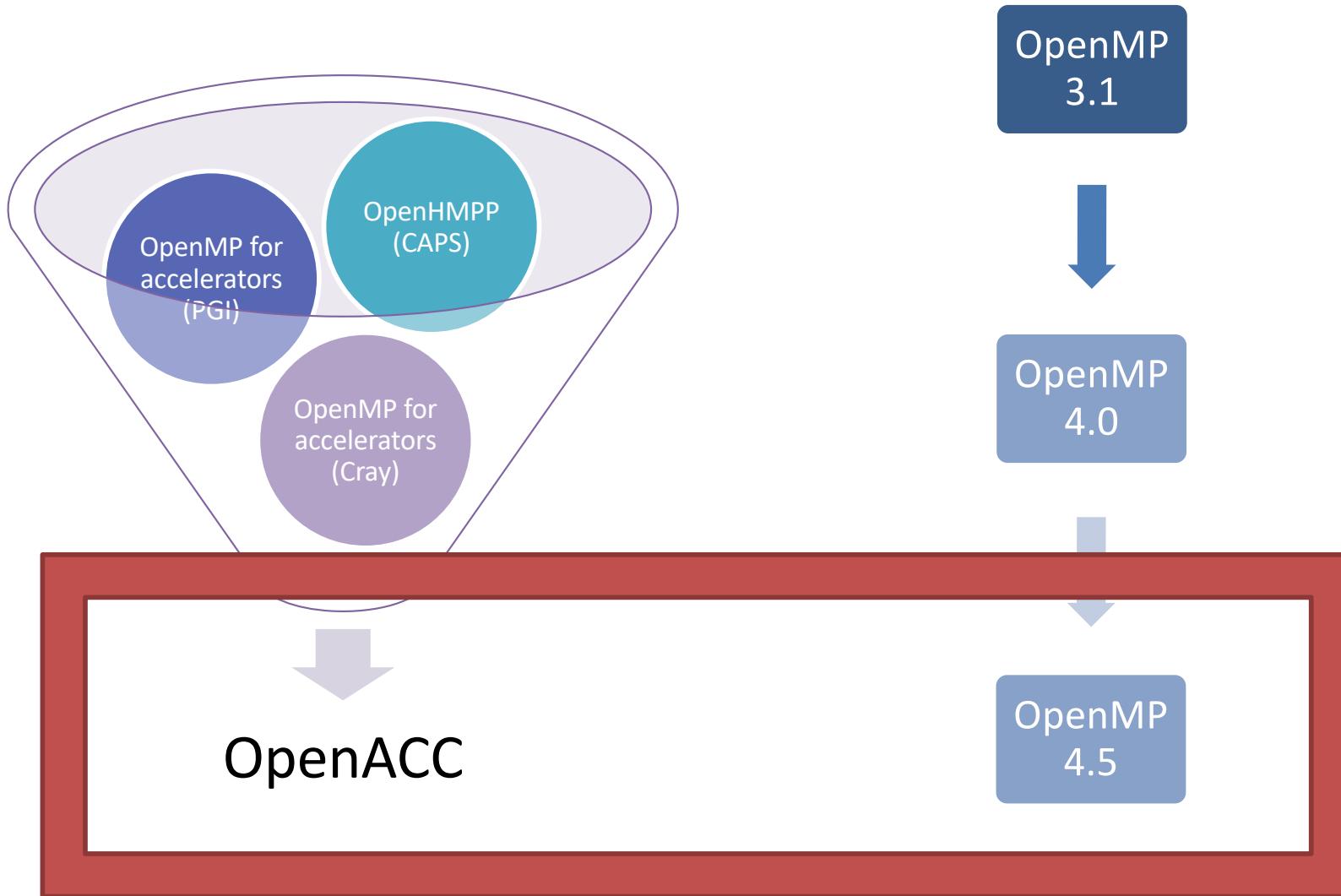
OpenCL

E usando BLAS?

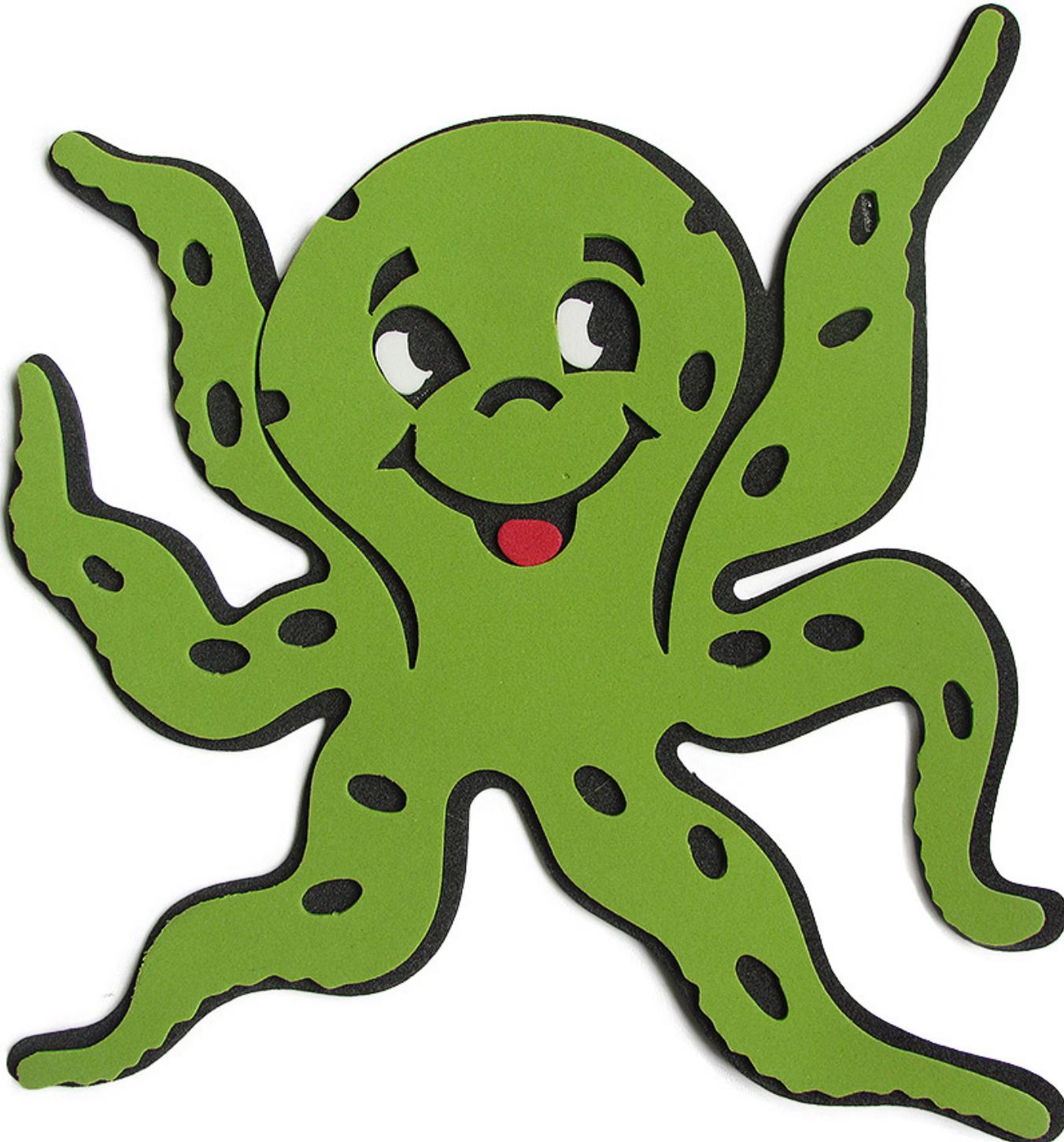
subroutine dgemm (TRANS_A, TRANS_B, M, N,
K, ALPHA, A, LDA, B, LDB, BETA, C, LDC)

(em Fortran!)

Quais padrões aceleram códigos em aceleradores?



- Organizaçõe
-
-
-
-
-
-
-



OpenACC e OpenMP target

- Ambas aceleram códigos por pragmas

```
#pragma omp|acc nome_diretiva [cláusula  
[,cláusula]...]  
    bloco estruturado de código
```

```
!$acc|omp nome_diretiva [cláusula [,cláusula]...]  
    bloco estruturado de código  
!$acc|omp end nome_diretiva
```

Exemplo: calculando PI

```
#include <stdio.h>
#define N 1000000000
int main(void) {
    double pi = 0.0f; long i;

    for (i=0; i<N; i++)
    {
        double t=(double) ((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi=%f\n",pi/N);
    return 0;
}
```

Exemplo: calculando PI com OpenMP

CPU

```
#include <stdio.h>
#define N 1000000000
int main(void) {
    double pi = 0.0f; long i;
#pragma omp parallel for reduction(+: pi)

    for (i=0; i<N; i++)
    {
        double t=(double) ((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi=%f\n",pi/N);
    return 0;
}
```

Exemplo: calculando PI com OpenACC

```
#include <stdio.h>
#define N 1000000000
int main(void) {
    double pi = 0.0f; long i;

#pragma acc parallel for reduction(+: pi)
    for (i=0; i<N; i++)
    {
        double t=(double) ((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi=%f\n",pi/N);
    return 0;
}
```

Exemplo: calculando PI com OpenMP target

```
#include <stdio.h>
#define N 1000000000
int main(void) {
    double pi = 0.0f; long i;

//#pragma omp target teams distribute parallel for reduction(+: pi)

    for (i=0; i<N; i++)
    {
        double t=(double) ((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi=%f\n",pi/N);
    return 0;
}
```

Exemplo: calculando PI

```
#include <stdio.h>
#define N 1000000000
int main(void) {
    double pi = 0.0f; long i;
    //#pragma omp parallel for reduction(+: pi)
    //#pragma omp target teams distribute parallel for reduction(+: pi)
    //#pragma acc  parallel for reduction(+: pi)
    for (i=0; i<N; i++)
    {
        double t=(double) ((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi=%f\n",pi/N);
    return 0;
}
```

As diretivas OpenACC

1. **parallels**: executa de forma paralela e íntegra o bloco no acc.
2. **kernels**: cria e computa kernels (região paralela distribuída) no acc.
3. **data**: alocação e movimentação de dados
4. **enter data**: alocação e movimentação de dados até o fim do programa
5. **exit data**: movimentação e desalocação de dados alocados no **enter data**
6. **host_data**: endereça no host dados do acc. (ponteiro do target)
7. **loop**: descreve paralelismo do laço no acc.
8. **cache**: especifica dados para cache do acc.
9. **declare**: aloca dados no acc.
10. **atomic**: assegura que o bloco deve ser executado atomicamente no acc.
11. **update**: atualiza dados no acc. e/ou no host
12. **wait**: espera finalização de operação assíncrona
13. **routine**: cria rotina com código do acc. e expõe nome para o host
14. (link, executable, API com outras arquiteturas, procedure calls...)

Diretivas OpenMP

- Crédito do próximo slide
 - Tom Scogland
 - Oscar Hernandez
 - NNSA – Exascale Computing Project
 - 28 de janeiro de 2017
 - <https://www.exascaleproject.org/wp-content/uploads/2017/05/OpenMP-4.5-and-Beyond-SOLLVE-part-21.pdf>

OpenMP 4.5 Device Constructs

- Execute code on a target device
 - **omp target** [clause[, clause], ...]
structured-block
 - **omp declare target**
[function-definitions-or-declarations]
- Manage the device data environment
 - **map** ([map-type:] list)
map-type := alloc | tofrom | to | from | release | delete
 - **omp target data** [clause[, clause], ...]
structured-block
 - **omp target enter/exit data** [clause[, clause], ...]
 - **omp target update** [clause[, clause], ...]
 - **omp declare target**
[variable-definitions-or-declarations]
- Parallelism & Workshare for devices
 - **omp teams** [clause[, clause], ...]
structured-block
 - **omp distribute** [clause[, clause], ...]
for-loops
- Device Runtime Support
 - void **omp_set_default_device**(int dev_num)
 - int **omp_get_default_device**(void)
 - int **omp_get_num_devices**(void)
 - int **omp_get_team_num**(void)
 - int **omp_is_initial_device**(void)
 - ...
- Environment variables
 - OMP_DEFAULT_DEVICE
 - OMP_THREAD_LIMIT

Correlações OpenMP OpenACC

- Em linhas bem gerais (e devidamente sem os óculos que trazem a definição e contraste à realidade dos padrões)
- `omp target teams distribute parallel -> acc parallel`
- `omp target data -> acc data`
- `teams/threads/simd -> gangs/workers/vector`

Esta aproximação funciona!

- Programa (feito pelo Jairo *from scratch*) denominado “Modelagem Fletcher”
- Ninho de 4 laços

```
for timestep {  
    for Z {  
        for Y {  
            for X {  
                .....  
            }  
        }  
    }  
}
```

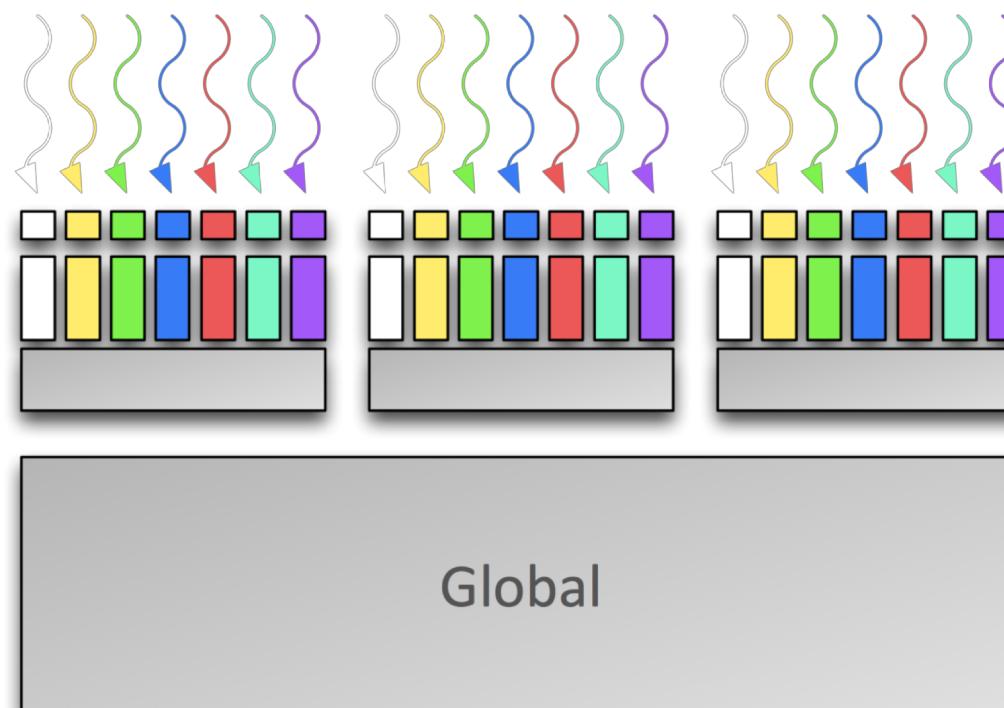
- Acelerado naïve com OpenACC
- Trocou-se os pragmas pelos “equivalentes” em OpenMP -> programa acelerado também!

Modelo de memória do OpenACC

- Memória do acelerador e do host **podem** estar separadas
 - Seja fisicamente, seja virtualmente, ou mesmo não estar (!!)
- Dados devem ser enviados e recebidos, alocados e dealocados
 - Mas estas operações são em grande maioria implícitas, o compilador trata isso (mas não é obrigatório!)
- Pode não existir coerência entre memória do host e acelerador
- Cópia da memória do host para a acelerador pode ser lenta
- No acelerador não há coerência entre threads
 - Race-conditions pode ocorrer
 - Programas que não tratam estes problemas são programas errados
- Alguns aceleradores possuem uma memória cache, rápida mas pequena
 - Ajuste fino do uso deste cache aumenta substancialmente o desempenho

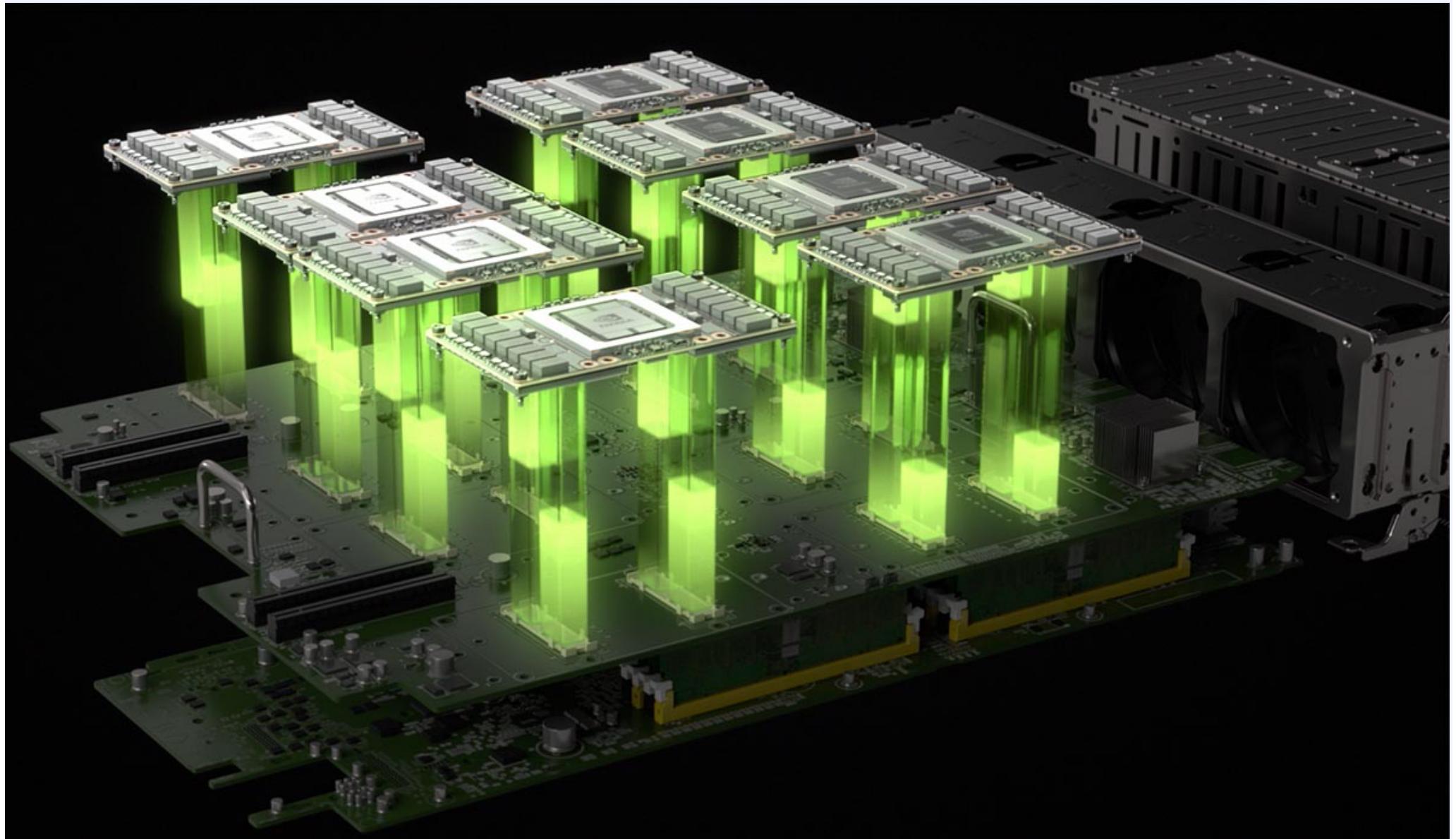
Hierarquia de memória

- Memória local de uma *thread*
- Memória compartilhada no bloco de *threads*
- Memória global entre blocos de *threads*



Fonte: Sarah Tariq, NVIDIA

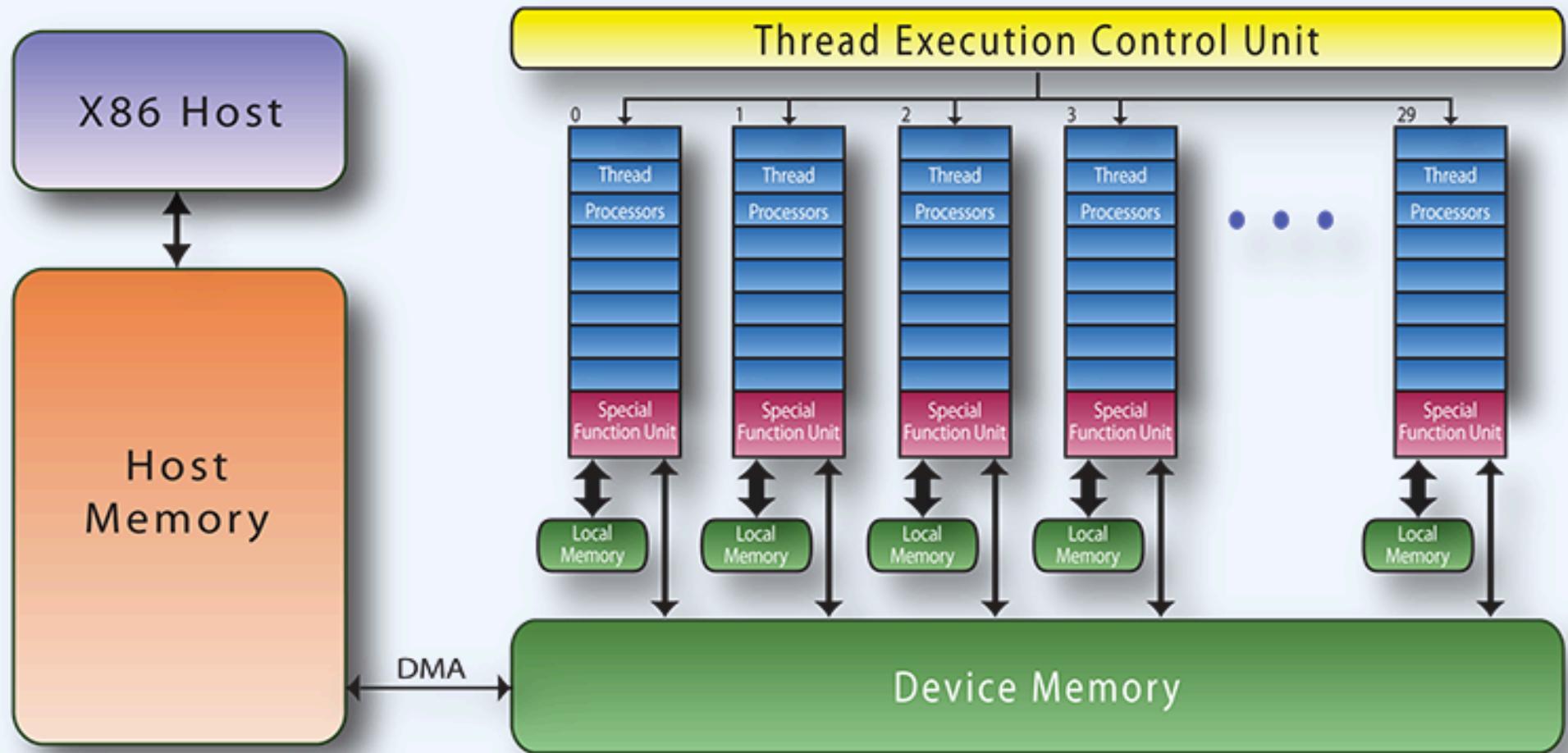
NVIDIA Architecture



Modelo de execução do OpenACC

- Grande parte do código é executado na host
- Regiões intensivas de cálculo são "offloaded", migrados para a host
 - Mas o programador deve explicitamente orientar o compilador, através das diretivas
- Trechos de código a serem acelerados criam “regiões computacionalmente intensivas”
 - Regiões paralelas, regiões de kernels, regiões seriais
- Mesmo em regiões intensivas o host também trabalhará
 - Cuida da execução, sincronismos, movimentação de dados...
- Um acelerador pode lançar paralelismo em *gangs*, cada *gang* possui um ou mais *workers* sendo que em um *worker* poderão ocorrer operações vetoriais denominadas *vector*
- Não fazer sincronismo!
- Pode existir “paralelismo aninhado”
- Também pode existir execução assíncrona

NVIDIA Architecture



The Portland Group®

Accelerator Overview Demo
©2009 The Portland Group, Inc.



Modelo de memória do OpenMP

- Muito mais complicado
 - Temos que entender o modelo para a CPU e depois para o target
- Basicamente cada thread tem uma memória
 - Elas podem não possuir coerência
 - O device target tem uma task atribuída, que por sua vez tem uma memória atribuída
 - Pode ou não estarem sincronizadas

Modelo de execução do OpenMP

- Também complicado
 - Também baseado no modelo de memória de CPU
 - “The OpenMP API uses the fork-join model of parallel execution” (primeira frase do item 1.3 do padrão) = diferente ao OpenACC

Limitações importantes

- Chamadas de rotinas em regiões aceleradas era um problema, solucionado com “acc routine”
 - Ou inline (manual ou automático)
- Variáveis derivadas de módulos (array de um elemento de um tipo em Fortran) não podem ser copiados para o acelerador
 - As vezes o compilador precisa saber o “formato” do *array*
 - Esta operação se chama “deep-copy” e o tamanho do *array* só é conhecido em “runtime”
 - Solução para *arrays* complicados: cópia de memória explícita
 - Solução promissora proximamente, pois o *deep-copy* parece estar no radar do padrão OpenACC (ver TR-16-1 de abril/2016)

Esse deep-copy...

- To allow for manual deep copy of data structures with pointers, new *attach* and *detach* behavior was added to the data clauses, new **attach** and **detach** clauses were added, and matching **acc_attach** and **acc_detach** runtime API routines were added; see Sections [2.6.3](#), [2.7.11-2.7.12](#) and [3.2.34-3.2.35](#).

1.11. Topics Deferred For a Future Revision

The following topics are under discussion for a future revision. Some of these are known to be important, while others will depend on feedback from users. Readers who have feedback or want to participate may post a message at the forum at www.openacc.org, or may send email to technical@openacc.org or feedback@openacc.org. No promises are made or implied that all these items will be available in the next revision.

- Support for attaching C/C++ pointers that point to an address past the end of a memory region.
- Full support for C and C++ structs and struct members, including pointer members.
- Full support for Fortran derived types and derived type members, including allocatable and pointer members.

Dicas

- Laços aninhados são os melhores para serem acelerados
 - Aproveitam a hierarquia de memória
 - Aproveitam os níveis de paralelismo do acelerador
- Sobreposição de comunicação com computação é possível e indicado
 - As regiões podem ser síncronas ou assíncronas
 - Ver diretiva “wait”

Dicas

- Iterações dos laços devem ser independentes
 - “Laços triangulares” não podem ser acelerados
- Ajude o compilador: use a chave “restrict” ou “independent” do C
- Compiladores podem se perder com o gerenciamento do que enviar/receber ao acc.
- Não utilizar aritmética de ponteiros
- Use memória contígua para arrays multidimensionais
 - E o percorra de acordo com sua construção na memória
 - C é diferente de Fortran!!!

Padrões semelhantes mas diferentes

- OpenACC
 - Somente sobre aceleração de códigos para aceleradores
 - Modelo de memória e execução estrito e enxuto
 - Descreve como é o código a ser acelerado e o compilador tem alto grau de liberdade
- OpenMP 4.5
 - Imprecisão na definição neste curso: este padrão nesta versão é muito mais que acelerar códigos em aceleradores
 - Modelo de memória é tanto para CPU como para target
 - Baseia-se em paralelismo por threads, e não aceleração
 - Diminui o grau de liberdade do compilador

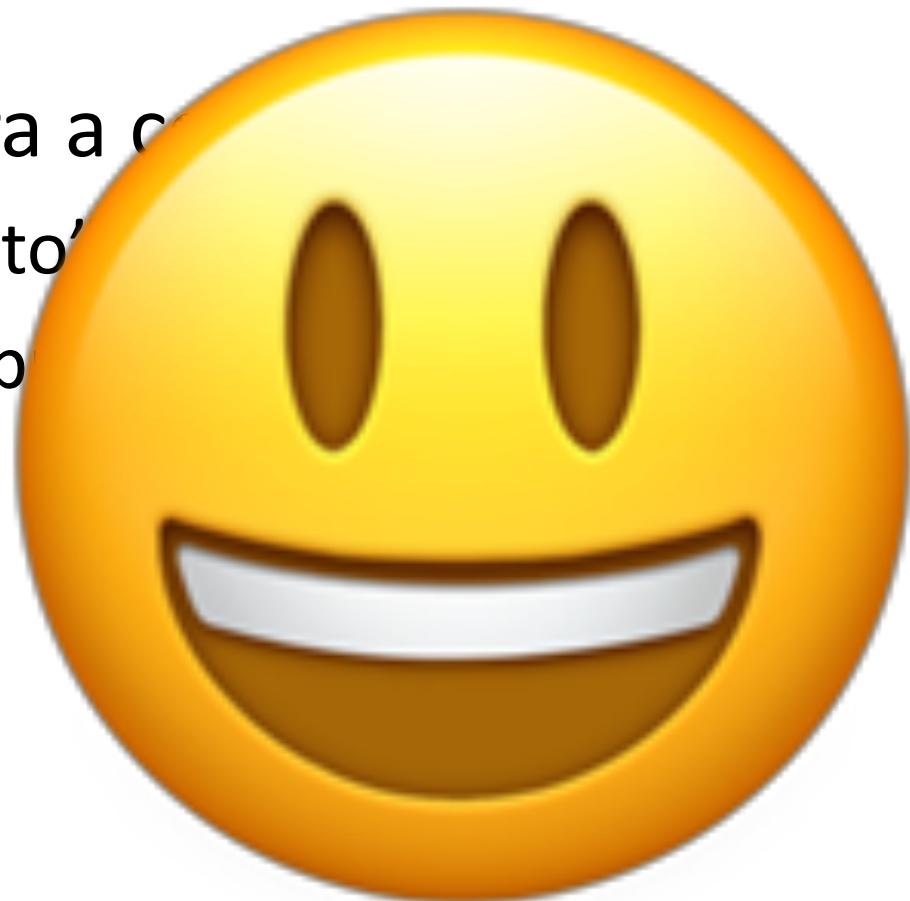
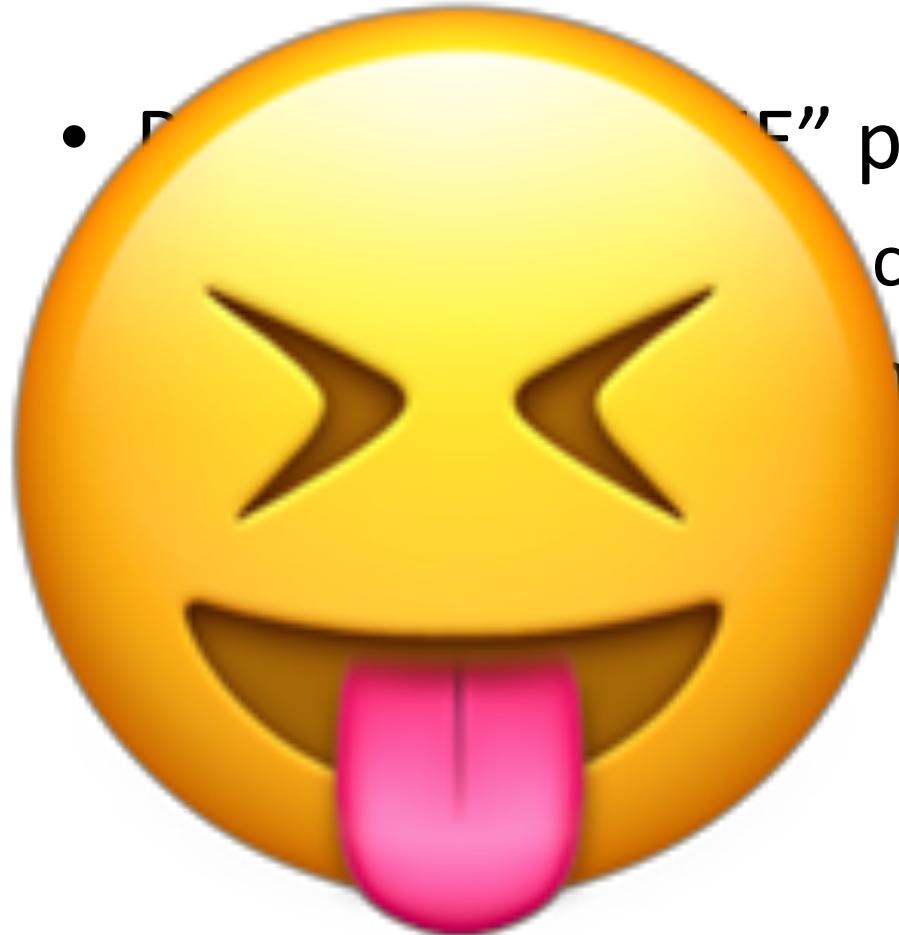
Parte 2: COMPILEDORES

Compilador PGI

- Ótimo compilador PAGO
- Não precisa do CUDA Toolkit para instalação, mas precisa dos drivers da Nvidia
- Referência em acelerar códigos
 - Tanto é que a NVIDIA comprou a PGI
- Linux, Windows, Mac OSX
- Precisa ser administrador para instalação
- Valores (fev/2017): US\$ 1.999 licença “flutuante” ou US\$ 1.399 licença “node-locked”
 - 50% de desconto para academia
- www.pgroup.com

Novidade boa

- P “E” para a c custo/n/p



Instalação do PGI (passo-a-passo)

- Ir no site
- Fazer o download do arquivo para a sua arquitetura (99% de ser Linux x86-64)
- Criar um diretório (ex, pgi-inst) e entrar nele
- Descompactar arquivo obtido
- Executar o script de instalação ./install

Instalação do PGI

- Quando perguntado, escolher "Single system install" (opção 1)
- O diretório de instalação deve ser um que seu usuário possa escrever (ex. /home/usuario/pgi)
- Ir confirmando, confirmando, confirmando
- Quando perguntado pela licença, digitar opção 4 (I'm not sure (quit now and re-run this script later,))
- Confirmar até terminar

Instalação do PGI

- Após instalação, setar seu PATH e seu LD_LIBRARY_PATH
- `export PATH=/home/usuario/pgi/linux86-64/2017/bin:$PATH`
- `export LD_LIBRARY_PATH=/home/usuario/pgi/linux86-64/2017/lib:$LD_LIBRARY_PATH`
- Testar (com os exemplos deste curso...)

Compilador Cray

- Pertencente ao Cray Linux Environment (CLE)
- ~~Responsável por aglutinar e publicar o padrão OpenACC~~

(era: o cara que fazia isso foi trabalhar na Nvidia!)

 - O compilador próprio tem suporte total ao padrão
 - Muito mais “amigável” no tratamento de regiões aceleradas do código
- Um compilador Cray só existe em um Cray

GCC+OpenACC (slide de 2015)

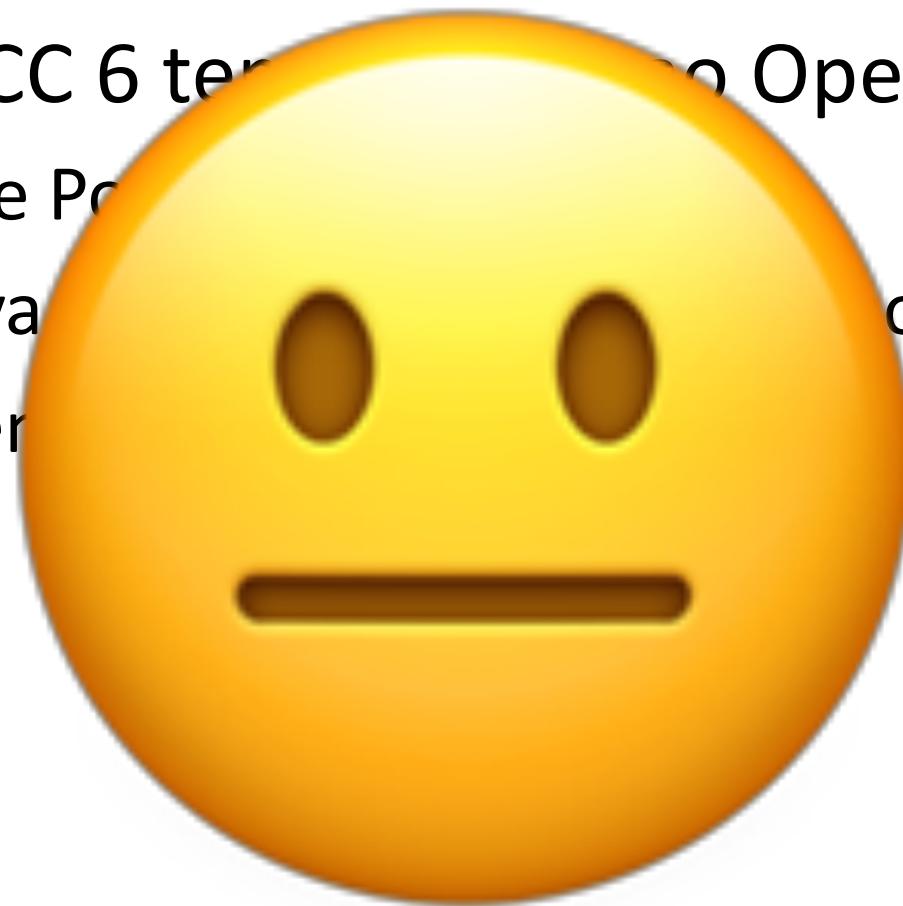
- PathScale (☺) é pago e não testei (☹)
- GCC (isso sim é uma boa notícia!)
 - A “Mentor ~~Embedded~~ Graphics” está fazendo esta herculana tarefa
 - Dizem que é previsto para o 5.0
 - *OpenACC support has been merged into GCC 5, but a handful of patches are still pending that are needed for nvptx offloading, so that's not yet functional.* (visto em <https://gcc.gnu.org/wiki/OpenACC>)

GCC+OpenACC

- Adivinha: eles conseguem ser integrados?
- Status: gcc 5.1.0 com OpenACC 1.0
 - Não tem o tipo `acc_declare`, `host_dat`
 - A diretiva `acc` não é implementada
 - Tem diversos erros
 - <http://bit.ly/1JLqfDw>

GCC+OpenACC

- Status: GCC 6 tem suporte ao OpenACC 2.0a
 - x86_64 e Power
 - A diretiva `acc` é suportada
 - E o desempenho?



- Créditos dos próximos slides:

Verónica G. Vergara Larrea, ORNL

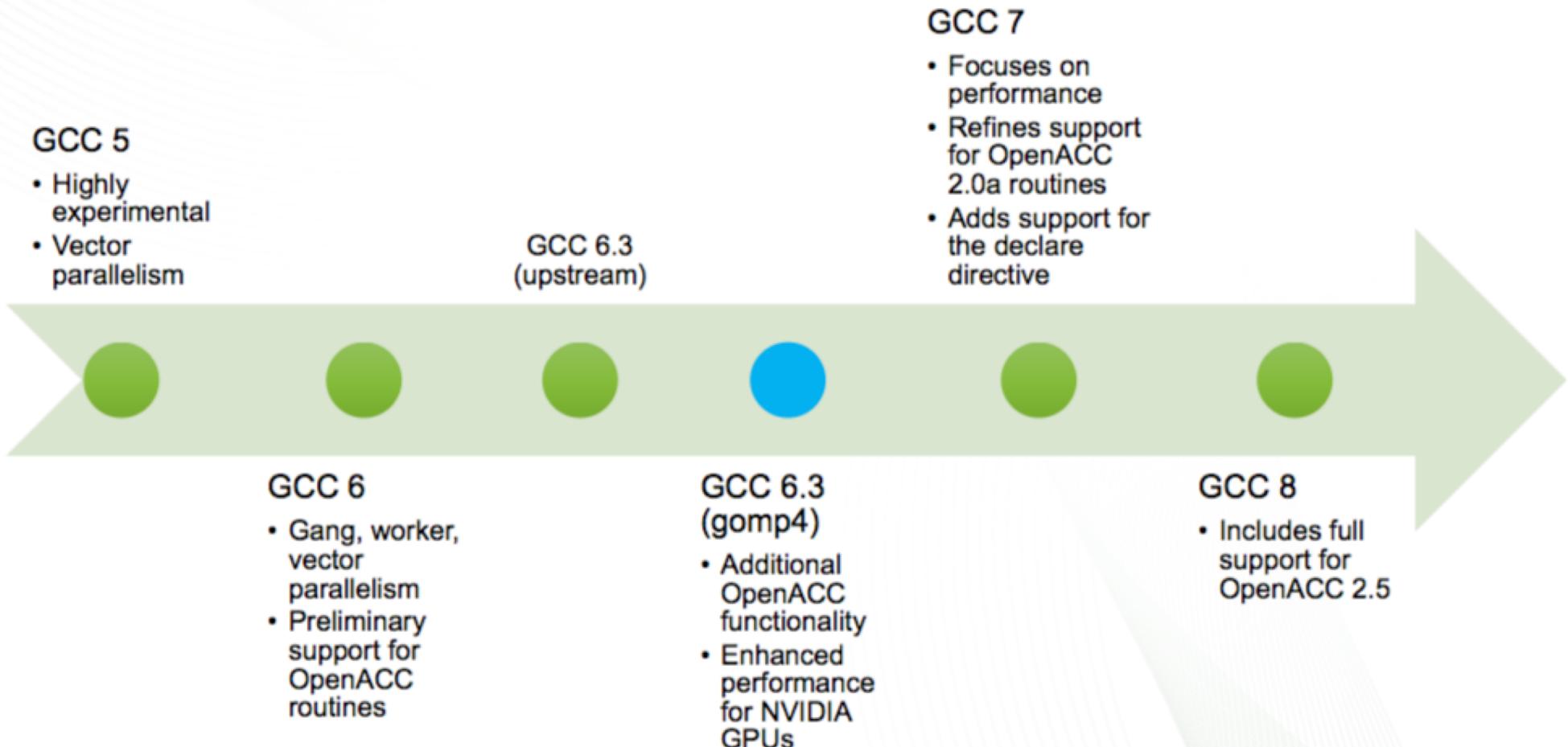
Wael R. Elwasif, ORNL

Oscar Hernandez, ORNL

Cesar Philippidis, Mentor Graphics

Randy Allen, Mentor Graphics

Evolution of GCC's OpenACC implementation



Porting Matrix Multiplication: Parallel

```
#pragma acc parallel
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        int t = 0;

        for (k = 0; k < n; k++)
            t += at(i, k, a) * at(k, j, b);

        at(i, j, c) = t;
    }
}
```

Porting Matrix Multiplication: Parallel Loop

```
#pragma acc parallel
#pragma acc loop
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        int t = 0;

        for (k = 0; k < n; k++)
            t += at(i, k, a) * at(k, j, b);

        at(i, j, c) = t;
    }
}
```

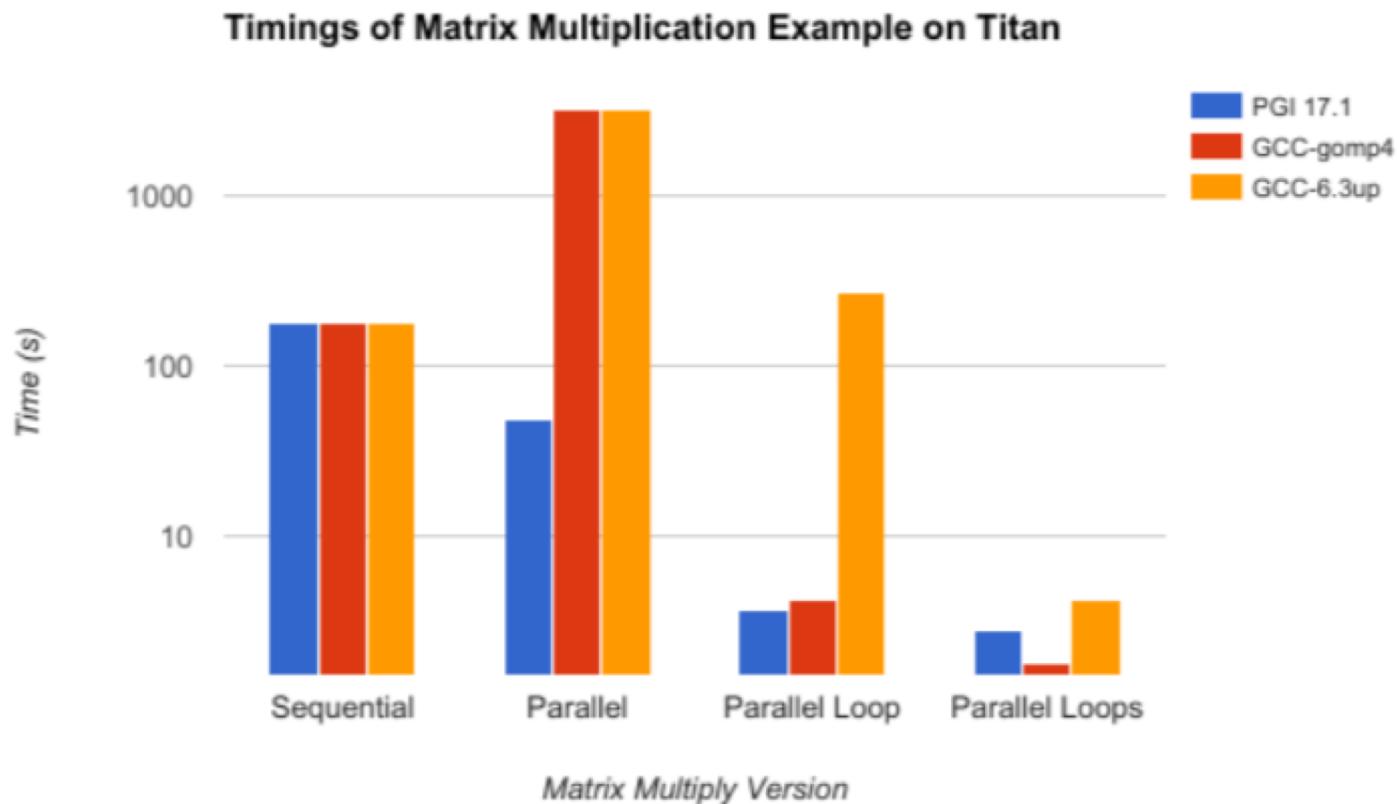
Porting Matrix Multiplication: Parallel Loops + Reductions

```
#pragma acc parallel present (a[0:n*n], \
b[0:n*n], c[0:n*n])
#pragma acc loop
for (i = 0; i < n; i++)
{
    #pragma acc loop
    for (j = 0; j < n; j++)
    {
        int t = 0;

        #pragma acc loop reduction (+:t)
        for (k = 0; k < n; k++)
            t += at(i, k, a) * at(k, j, b);

        at(i, j, c) = t;
    }
}
```

Porting Matrix Multiplication



Conclusions

- GCC's OpenACC implementation is now available with partial support for OpenACC v2.0a
 - Mentor Graphics public GCC branch gomp-4_0-branch has the latest updates
- GCC-gomp4 can in some cases outperform more mature implementations.
 - As was the case with the SPEC ACCEL 304.olbm benchmark
 - Overall, GCC is ~47% slower than PGI for SPEC ACCEL measured estimates
- Known limitations of the implementation reduce the number of tests available for the evaluation

Conclusions (cont'd)

- For portability, OpenACC implementations should support many targets
 - e.g., PGI achieves good performance on both GPU-based and manycore-based systems
 - To compare performance, support for additional architectures is needed in GCC's OpenACC implementation
- An open source implementation is useful to expand the adoption of OpenACC
- Many of the benchmarks available have not been recently updated
 - Community involvement could improve and encourage updates to benchmarks

Outros compiladores (2017)

- accULL: Universidad de La Laguna/EPCC
 - Funciona, compila códigos simples, é um “source to source”, convertendo trecho em OpenACC para CUDA, usa Python no meio do caminho...
- Omni: RIKEN/University of Tsukuba (Japão)
 - Melhorou bastante com relação a 2015
 - Teste com o pi: 0,787 segundos (excelente!)
 - Não foram feitos mais testes...

Outros compiladores

- RoseACC: University of Delaware/LLNL
 - Tem que fazer fork do github, outro source-to-source, não testei
- OpenUH: University of Houston
 - Beeeeeem complicando para instalar, é gigante, tem suporte alpha a OpenACC mas suporta Coarray Fortran, existe a anos
- OpenARC: Oak Ridge Nat. Lab.
 - Bastante científico, quem quiser se aventurar...

E o OpenMP target?

- GCC 7.3.0 tem suporte *parcial*
 - Mas suporta o OpenACC também
 - É um faz-tudo
- CLANG
 - É o que vamos ~~usar~~ (desculpem: ainda não...)
 - Como compila?

SPACK!

- spack.io



- git clone <https://github.com/spack/spack.git>
- source spack/share/spack/setup-env.sh
- spack install lmod (espere...)
- source \$(spack location -i lmod)/lmod/lmod/init/bash

Compilar o LLVM via SPACK

- spack install llvm (espere MUITO...)
- module load llvm-<tab> (ou veja o nome do módulo antes)
- Vai dar um Warning de possível lentidão devido a falta de BCLIB (bytecode library). Solução: compilar NOVAMENTE (sim, novamente outra vez!) o LLVM

... NA MÃO

Compilando o LLVM

- Criar um diretório para compilação (build)
- Pré-requisitos
 - UBUNTU NOVO (ou algo que você saiba mexer)
 - CUDA toolkit + drivers NVIDIA
 - Pacotes UBUNTU libelf-dev libffi-dev ninja cmake
 - 45 gigas de disco (ISSO MESMO!!!)
 - Muita paciência

LLVM na mão

- Baixar os pacotes do site do llvm.org
 - Do próprio llvm
 - Do CLANG (se vai chamar `cfe-alguma_coisa.tar.xz`)
 - Do OpenMP
- Descompactar o llvm
 - DICA: `tar xf nome_arquivo.tar.xz` descompacta!
- Descompactar o clang dentro do diretório `llvm-alguma_coisa/tools`
- Descompactar o OpenMP dentro do diretório `llvm-alguma_coisa/projects`
- Criar um diretório NO MESMO NÍVEL do llvm chamado build

LLVM na mão

- A linha mágica de três dias de engenharia reversa para encontrar

```
cmake -DCMAKE_INSTALL_PREFIX=diretório_completo_para_instalação \
-DLLVM_TARGETS_TO_BUILD="X86;NVPTX" \
-G Ninja -DLLVM_ENABLE_ASSERTIONS=ON -DLLVM_ENABLE_BACKTRACES=ON \
-DLLVM_ENABLE_WERROR=OFF -DBUILD_SHARED_LIBS=OFF \
-DLLVM_ENABLE_RTTI=ON \
-DLIBOMPARGET_NVPTX_COMPUTE_CAPABILITIES=35 \
-DCLANG_OPENMP_NVPTX_DEFAULT_ARCH=sm_35 \
-DLIBOMPARGET_NVPTX_ENABLE_BCLIB=true \
-DOPENMP_ENABLE_LIBOMPARGET=ON \
../llvm-alguma_coisa
```

(trocar 35 pelo Compute Capability da sua GPU)
(diretório_de_instalação deve ser diferente do build e dos sources)

LLVM na mão

- ninja (vá tomar vários cafés)
- ninja install (um café basta)
- Alguns exports no bash:

```
RAIZ=lugar_onde_esta_o_diretório_instalação
```

```
export CPATH=$RAIZ/include:$CPATH
```

```
export LD_LIBRARY_PATH=$RAIZ/lib:$LD_LIBRARY_PATH
```

```
export LIBRARY_PATH=$RAIZ/lib:$LIBRARY_PATH
```

```
export PATH=$RAIZ/bin:$PATH
```

```
export LIBOMP_LIBS=$RAIZ/lib
```

```
export OMPTARGET_LIBS=$RAIZ/lib
```

Brinque bastante com LLVM

- Compilação de programas com offload:

```
clang -fopenmp -fopenmp-targets=nvptx64-nvidia-cuda arquivo.c
```

Ah, CLANG gosta também de CUDA (arquivo.cu)!

(shhh... no GCC se compila OpenMP offload assim)

```
gcc -fopenmp laplace2d.c -foffload=-lm
```

(se quiser se aventurar:

<https://kristerw.blogspot.com.br/2017/04/building-gcc-with-support-for-nvidia.html>)

OpenACC OpenMP compiler matrix

OpenACC

Nome	Offload CPU	Offload GPU	Offload Phi
GCC 7.3.0	⚠		
PGI 18.10	○	○	
INTEL 2018			
CLANG YKT			
CLANG TRUNK			

OpenMP target

Nome	Offload CPU	Offload GPU	Offload Phi
GCC 7.3.0	⚠		
PGI 17.10			
INTEL 2018	○		○
CLANG YKT			(libiomp5)
CLANG TRUNK			⚠



= permite uso em produção



Parte 3: PRÉ-PRÁTICA

- Ambiente
- Flags de compilação e mensagens
- Execução do programa
- Medição de tempo total e por região acelerada

Ambiente

- Não temos SDumont! Instalar o PGI nas máquinas locais (faz parte do treinamento!)
- Copiar os arquivos do curso
`cp ~professor/MC07-SD/MC07-SD-sdumont_fev2019.tar.gz .`
- Descompacte-o!

Quem quiser levar pra casa

exaflop.com.br/MC07-SD-sdumont_fev2019.tar.gz

(apagarei em uma semana!)

Entendendo o pacote

• MC07-SD-sdumont.pdf	Esta apresentação
• e1-pi.c	Cálculo do PI para acelerar
• e2-laplace2d.c	Jacobi para acelerar
• e3-laplace2d.c	Jacobi para otimizar
• e4-laplace2d.c	Jacobi para OpenMP target
• roda-omp.srm	Arquivo de submissão CPU
• roda-acc.srm	Arquivo de submissão GPU
• solucao	Diretório de soluções
• - s1-pi.c	Solução do e1
• - s2-laplace2d.c	Solução do e2
• - s3-laplace2d.c	Solução do e3
• - s4-laplace2d.c	Solução do e4

Parte 3: PRÁTICA

Hands-on!

Seremos tendenciosos!

- Conceitos básicos com OpenACC + PGI
 - Movimentação de dados
 - Chaves de compilação
 - Seções aceleradas
 - Conceitos de paralelismo
- ~~Maior tempo neste caso~~
 - ~~Mas visitaremos OpenMP + CLANG e OpenMP + intel 2018 no fim!~~

Exercício #1 (faremos juntos!): e1-pi.c

- Objetivo: compilar, executar e medir tempo de execução do e1-pi para comparar execução sequencial, OpenMP CPU e OpenACC
- Conceitos: paralelismo, speed-up, medição confiável de tempo
- Anotar a fórmula:

$$S_p = \frac{T_1}{T_p} \quad \begin{array}{l} S \Rightarrow \text{Speed-up} \\ p \Rightarrow \text{número de processadores} \\ T \Rightarrow \text{Tempo} \end{array}$$

Passos!

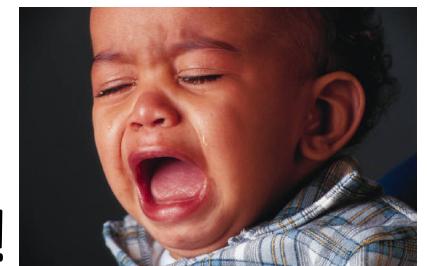
- Abrir o arquivo com algum editor
- Entender o laço
 - Ele pode ser executado em paralelo?
- Compilar!
- Executar e cronometrar
- Anotar
- Calcular speed-up

O e1-pi.c

```
#include <stdio.h>
#include <omp.h>
#define N 1000000000
int main(void) {
    double pi = 0.0f; long i;
    printf("Numero de processadores = %d\n", omp_get_max_threads());
    #pragma omp parallel for reduction(+: pi) /* OpenMP CPU */
    #pragma acc parallel loop reduction(+: pi) /* OpenACC */
    for (i=0; i<N; i++)
    {
        double t=(double) ((i+0.5)/N);
        pi += 4.0/(1.0+t*t);
    }
    printf("pi=%f\n",pi/N);
    return 0;
}
```

Compilação “na mão”

- Receita
 - Compilador: “ pgc++ ”
 - Chave para criação do executável: “ –o e1-pi ”
 - Chave para ligar OpenMP (independe de qual): “ –mp ”
 - Chave para ligar OpenACC: “ –acc ”
 - Chave para escolher o acelerador: “ –ta=nvidia ”
 - Saber o que ele está fazendo: “ –Minfo ”
- ATENÇÃO: não utilize –mp e –acc junto!
 - A máquina explodirá e criará um buraco-negro
- Bonus: poderemos fazer uso dos dois no fim!



Mas tem Makefile!!!

- Digite “make” e as opções mostram o que pode ser feito
- Alguns tem OpenMP CPU implementado, outros não, mas se quiser implementar OpenMP CPU fiquem a vontade
- make clean limpa tudo!

Medir tempo

- Simples, muito simples

/usr/bin/time -p ./e1-pi-omp (o roda.srm tem!)

- Dica: com a chave “–mp” para controlar o número de processadores use a variável OMP_NUM_THREADS
 - Neste curso se troca o número de threads no rodar-omp.srm na linha 5 opção --cpus-per-task

/usr/bin/time -p OMP_NUM_THREADS=4 ./e1-pi-omp

Medir tempo com acelerador

- Também é muito simples
 - Igual ao OpenMP CPU
- Para assegurar que o acelerador está sendo usado modifique a variável ACC_NOTIFY

```
/usr/bin/time -p ACC_NOTIFY=1 ./e1-pi-acc
```

Resultados

- São compatíveis com o apresentado?

Tipo	Sem diretivas	OpenMP(4)	Acelerador
Tempo (s)	6,118	2,238	0,351
Ganho (vezes)	---	2,73	17,43

Diretiva utilizada: parallel

- A `parallel` criou uma seção paralela no bloco estruturado seguinte, lançando vários *gangs*
 - Todas as *threads* executam a mesma coisa redundantemente!
- Lançou vários *gangs* para as iterações do laço
- Como havia a outra diretiva “`loop`” logo a seguir quebrou o espaço de índices em pedaços e os distribuiu nos *gangs*
- A cláusula “`reduction`” realizou a soma dos *gangs*

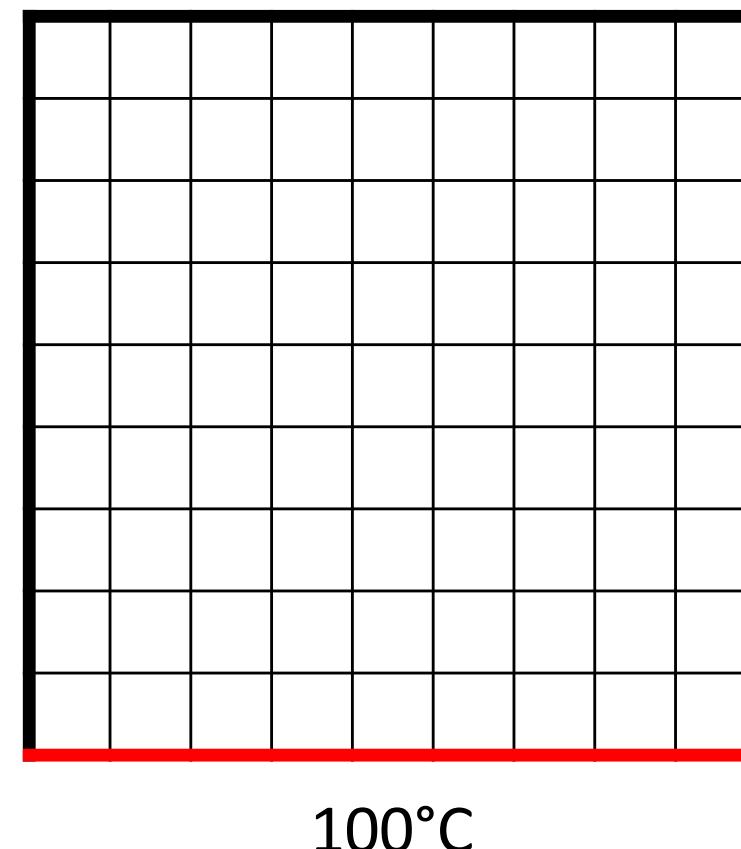
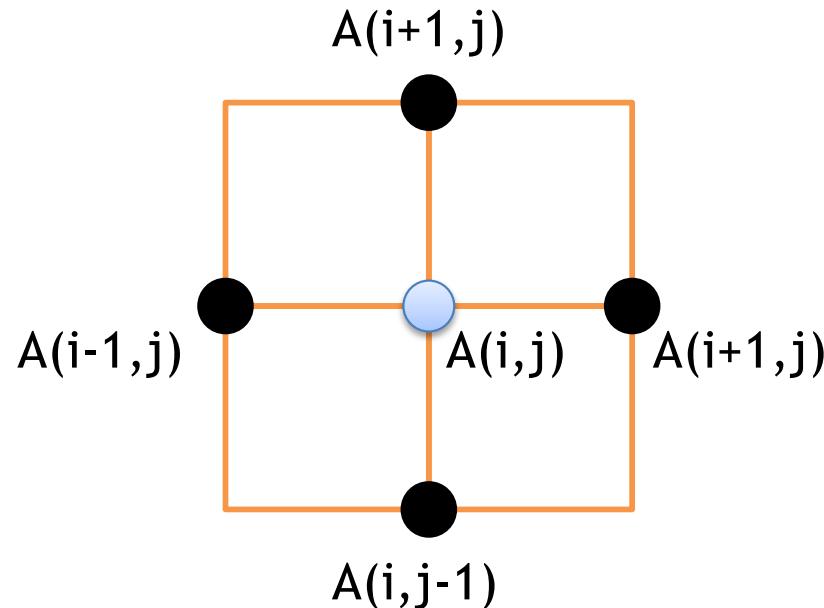
Exercício #2: e2-laplace2d.c

- Objetivo é acelerar um programa mais complexo, com dois laços e um “while”
- Ele já está paralelizado com OpenMP CPU
- Tarefa: encontrar a diretiva e implementar
- ATENÇÃO: o programa deve parar cerca da iteração 2420
 - Se continuar até a 3000 = programa errado

O problema

- Converge iterativamente para um valor a partir da média dos pontos vizinhos
- Exemplo: solução da

Equação de Laplace



Diretiva a utilizar: kernels

- A “kernels” cria um ou mais *kernels* no acelerador para executar laços eficientemente
- Utiliza bem os graus de paralelismo existentes
- É rápido, muito inteligente mas não tem as mesmas operações do “parallel”
 - VER NO PADRÃO AS DIFERENÇAS
- Respondendo a pergunta: pragma kernels é a essência do OpenACC -> ela é descritiva

Diretiva kernels ("foto" do RefGuide)

Kernels Construct

A **kernels** construct surrounds loops to be executed on the device, typically as a sequence of kernel operations.

C/C++

```
#pragma acc kernels [clause [,] clause]... new-line
{structured block}
```

FORTAN

```
!$acc kernels [clause [,] clause]...
structured block
 !$acc end kernels
```

Compute Construct and Data clauses are also allowed; data clauses on the kernels construct modify the structured reference counts for the associated data.

Como rodar

- Compile para OpenMP (use o make!)
- Altere o rodar-omp.srm para refletir o nome do executável que foi criado (verifique se o número de threads -cpus-per-task é 56)
- sbatch rodar-omp.srm
- Espere...
- Veja o arquivo de log do slurm...
- Anote o tempo com OpenMP cpu
- Atenção: execução sequencial demora MUITO

Como rodar

- Altere o código incluindo a diretiva kernels
- Compile para OpenACC
- Altere o rodar-acc.srm para refletir o nome do executável que foi criado (verifique se o número de threads -cpus-per-task é 1)
- sbatch rodar-acc.srm
- Espere...
- Veja o arquivo de log do slurm...
- Anote o tempo com OpenACC

Exercício

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;  
    #pragma omp parallel for shared(Anew, A) reduction(max:error)  
  
    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, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
    #pragma omp parallel for shared(Anew, A)  
  
    for( int j = 1; j < n-1; j++) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
    iter++;  
}
```

Solução

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;  
    #pragma omp parallel for shared(Anew, A) reduction(max:error)  
    #pragma acc kernels  
    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, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
    #pragma omp parallel for shared(Anew, A)  
    #pragma acc kernels  
    for( int j = 1; j < n-1; j++ ) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
    iter++;  
}
```

Que péssimo desempenho!

- Compare os tempos com e sem acelerador
- Onde está o problema?

Compilador fez o melhor que pode

- Mas ele NÃO pode desrespeitar o código

1. Enquanto condicional do while for satisfeita...

2. copia A para a placa

3. Computa primeiro laço

4. Copia Anew para CPU

5. Copia Anew para placa

6. Computa segundo laço

7. Copia A para a CPU

8. Volta para 1.

```
while ( error > tol && iter < iter_max ) {
    error=0.0;
#pragma omp parallel for shared(Anew, A) reduction(max:error)
#pragma acc kernels
    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, abs(Anew[j][i] - A[j][i]));
        }
    }
#pragma omp parallel for shared(Anew, A)
#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[i][j];
        }
    }
    iter++;
}
```

Exercício #3: e3-laplace2d.c

- Objetivo é otimizar a execução do exercício #2
- Ele já está acelerado
- Verifique a problemática
 - Entenda “no código” e “no algoritmo” o problema
- Este exercício pode ser melhorado ainda mais

Diretivas a utilizar: data

- A diretiva “data” diz ao compilador, com suas cláusulas
 - copy(vetor): Copie um vetor para o acelerador e o traga de volta no fim
 - copyin(vetor): Copie um vetor para o acelerador
 - copyout(vetor): Copie um vetor para o *host*
 - create(vetor): Criar um vetor no acelerador mas não o inicializa com dados

(lembra:

```
#pragma acc nome_diretiva [cláusula [,cláusula]...]
    bloco estruturado de código )
```

Perguntas interessantes

- A matriz A precisa estar no acelerador?
 - Ela está criada lá?
 - Precisa ser inicializada?
 - Precisa dela no fim dos cálculos?
- A matriz Anew serve para quê?
 - Precisa dela no acelerador?
 - Precisa dela no host?
- Traduzir a resposta destas perguntas em operações de cópia das matrizes “dê” e “para” o acelerador

Solução

```
while ( error > tol && iter < iter_max ) {  
    error=0.0;  
    #pragma omp parallel for shared(Anew, A)  
    #pragma acc kernels  
    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, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
    #pragma omp parallel for shared(Anew, A)  
    #pragma acc kernels  
    for( int j = 1; j < n-1; j++ ) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
    iter++;  
}
```

Solução

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.0;
#pragma omp parallel for shared(Anew, A)
#pragma acc kernels
    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, abs(Anew[j][i] - A[j][i]));
        }
    }
#pragma omp parallel for shared(Anew, A)
#pragma acc kernels
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }
    iter++;
}
```

Possível melhora!

- Tirar os dois kernels
- Colocar um “parallel” com reduction
 - Será? Juro que não sei!!!
- Especificar os loops
- Mudar o número de gangs, workers e vectors
- Usar a diretiva cache
 - Uso da shared memory

Exercício #4: OpenMP target

- Trocar o e3-laplace2d.c de OpenACC para OpenMP com target
- Forneço as linhas de pragmas

```
#pragma omp target data map(to:Anew) map(tofrom:A)
```

```
#pragma omp target teams distribute parallel for collapse(2)  
reduction(max:error) map(tofrom:error)
```

```
#pragma omp target teams distribute parallel for collapse(2)
```

O que fazer?

- Rodar
- Medir tempo
- Comparar

Exercício #4a: rodar no Xeon Phi

- Troque o compilador para o intel
- Troque a fila para “??????” no rodar-acc.srm
- Medir tempo, comparar
- Em tempo: o ambiente de compilação não está funcionando, infelizmente
 - Mas se você tiver acesso a um que funcione
 - RODE E ME ESCREVA COM O RESULTADO!!!

Exercício #5: multi-GPU

- Faça um programa “do zero” que apresente (bom) speed-up usando múltiplas GPUs
- Não precisa mostrar resultado
- Dicas
 - OpenACC gera muito paralelismo
 - Tudo fica rápido
 - A GPU é muito rápida
 - Pense em overhead de outras operações

Sugestão

- SAXPY
- Aloque arrays múltiplos de 2, 4 e 8
- Pense nos overheads
- Acho que precisa mexer no rodar-acc.sqm
 - Verificar na documentação!
- Faça uma curva de speed-up!

Exercício #6 (pra quem puder fazer)

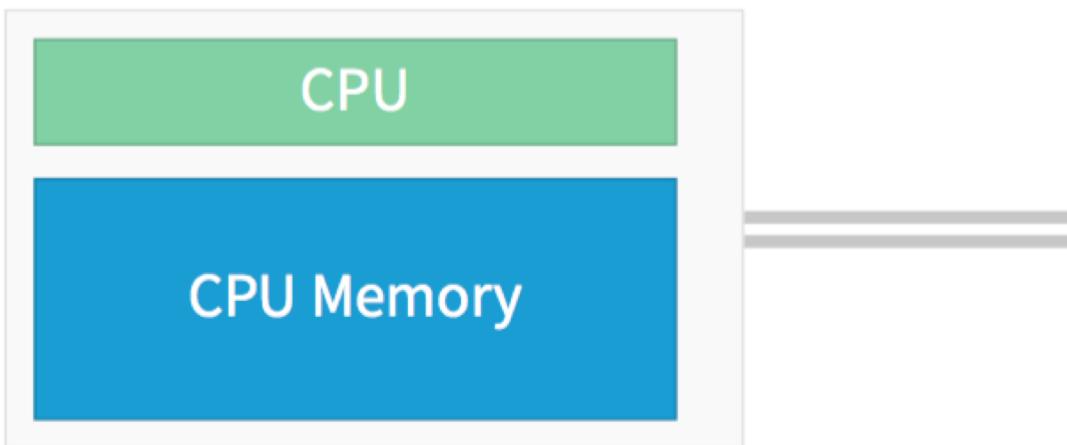
- Utilizar o modo “managed” do PGI 17.10
- Quem tem uma tesla V100 ai?
- (os próximos slides contam o futuro)
- Crédito dos próximos slides: Andreas Herten



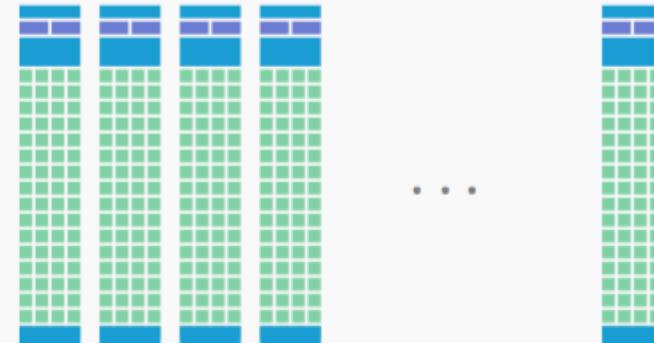
GPU Memory Spaces

Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses



Scheduler



Interconnect

L2

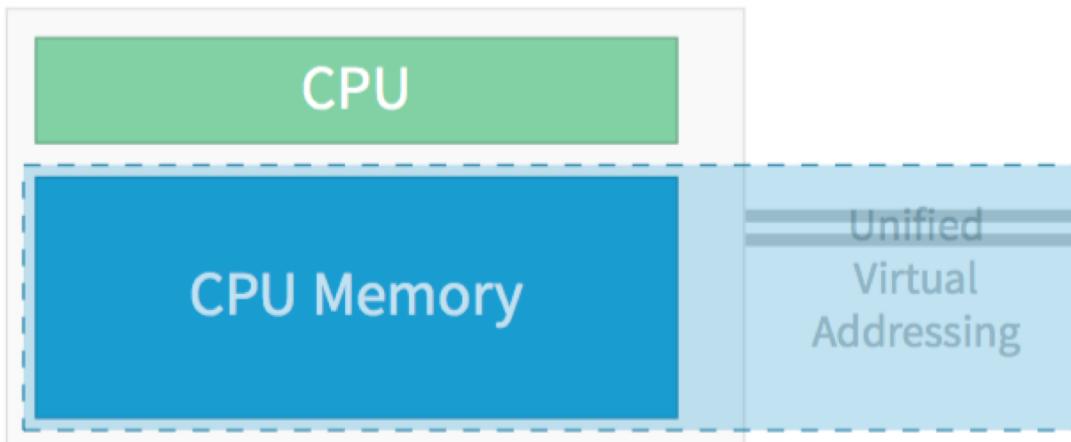


GPU Memory Spaces

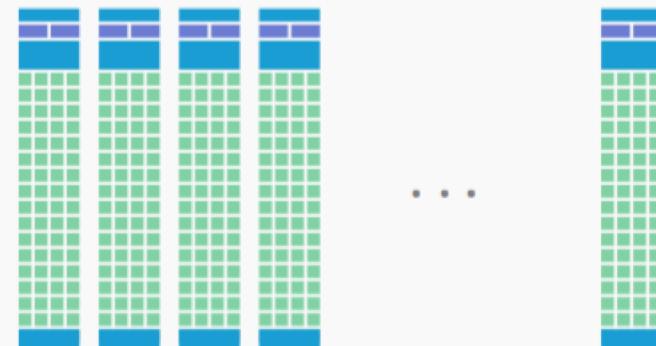
Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual



Scheduler



Interconnect

L2

DRAM

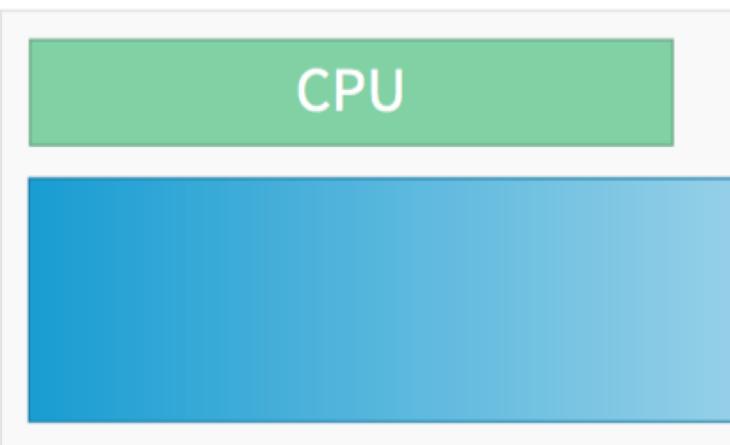
GPU Memory Spaces

Location, location, location

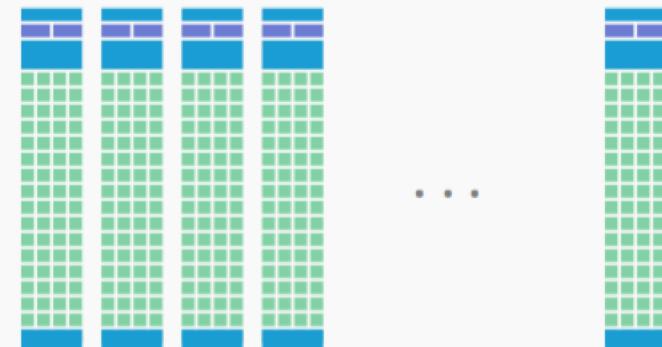
At the Beginning CPU and GPU memory very distinct, own addresses

CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual

CUDA 6.0 Unified Memory*: Data copy by driver, but whole data at once (Kepler)



Scheduler



Interconnect

L2

Unified
Memory

GPU Memory Spaces

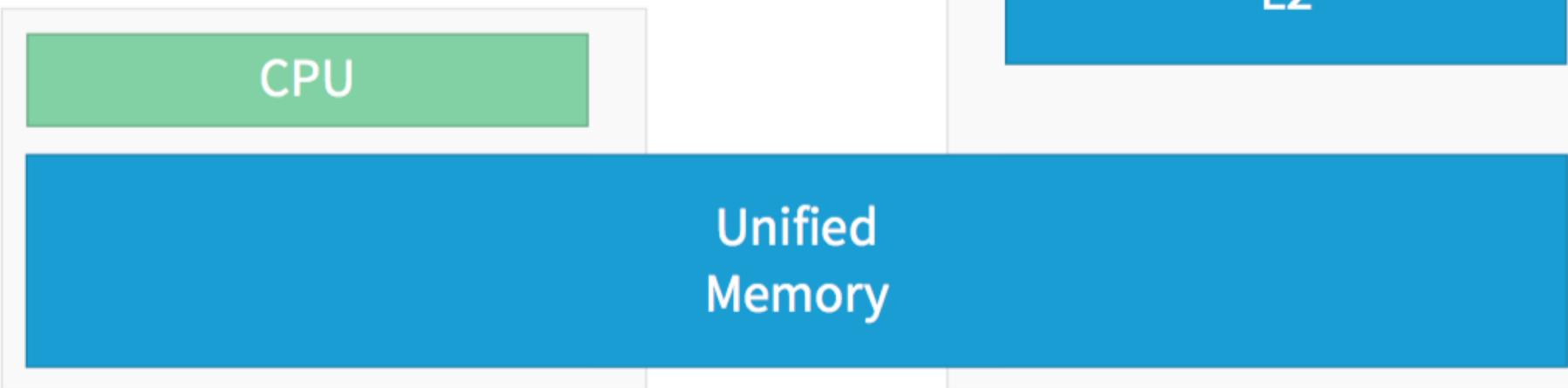
Location, location, location

At the Beginning CPU and GPU memory very distinct, own addresses

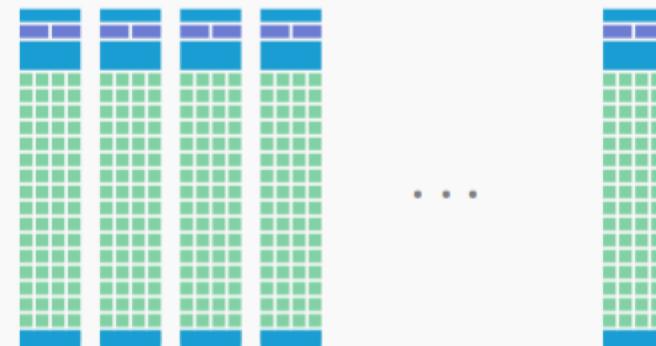
CUDA 4.0 Unified Virtual Addressing: pointer from same address pool, but data copy manual

CUDA 6.0 Unified Memory*: Data copy by driver, but whole data at once (Kepler)

CUDA 8.0 Unified Memory (truly): Data copy by driver, page faults on-demand initiate data migrations (Pascal)



Scheduler



Interconnect

L2

NVIDIA Tesla V100 - GPU computing processor - Tesla V100 - 16 GB

Price:

\$8,720.99

Availability:

Call For Availability

Mfr #: 900-2G500-0000-
000

UNSPSC #: 43201401

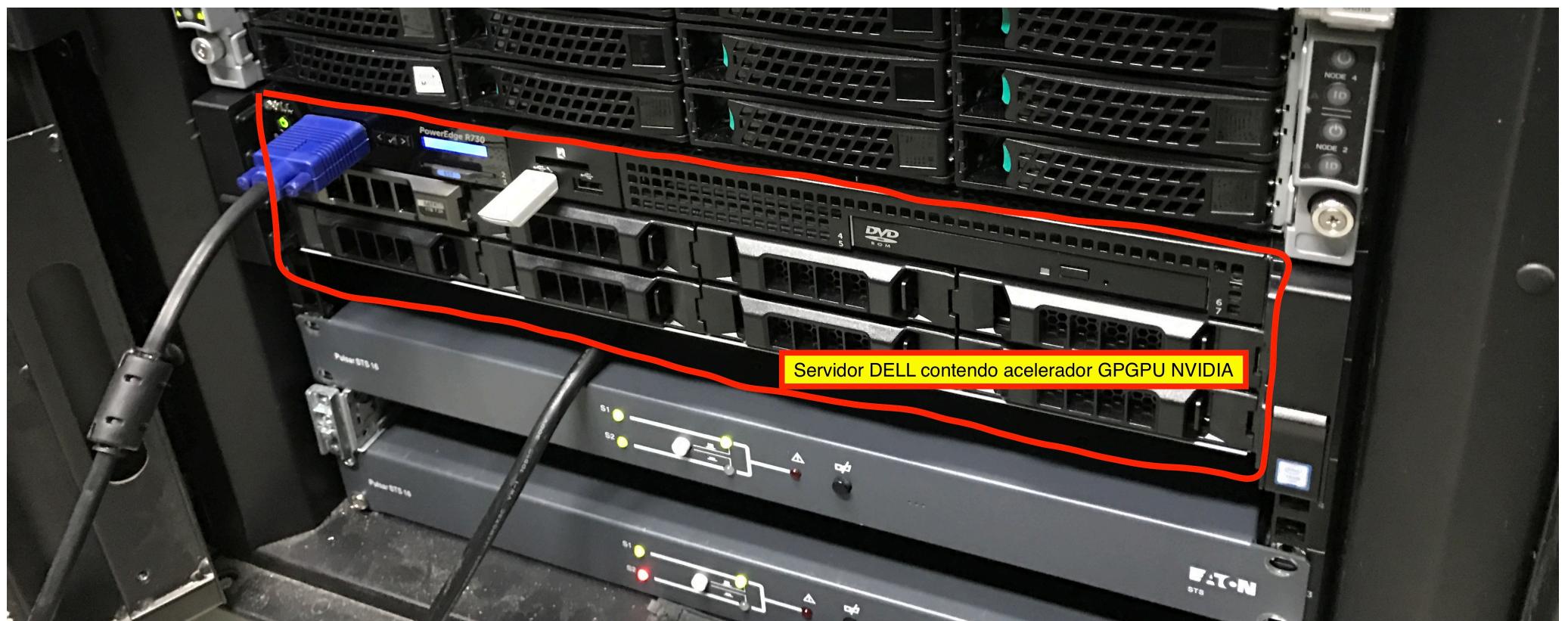
Item #: 005184598

[Add to Shopping List](#)



Não esqueça do servidor

- Ela não tem cooler
- ...



Obrigado!!!

www.exaflop.com.br

pedro.lopes@exaflop.com.br

