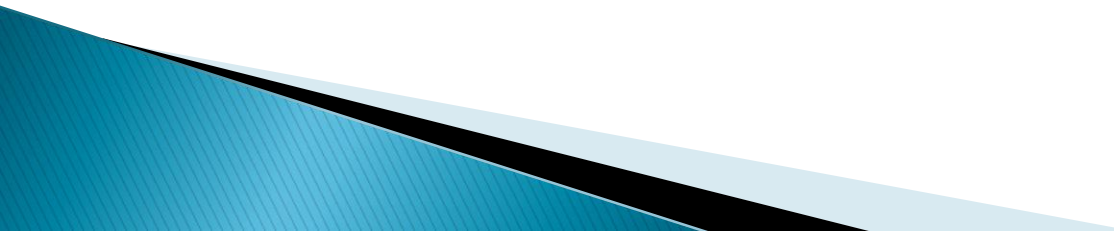


# Modelagem Atmosférica usando GPU

Pedro da Silva Peixoto (MAP-IME-USP)



# Sumário

- ▶ Introdução
  - ▶ Portando modelos atmosféricos existentes para a GPU
  - ▶ Autoparalelismo para GPU em modelo atmosféricos
  - ▶ Conclusões
- 

# INTRODUÇÃO

# O Problema

- ▶ Resolver as equações envolvidas na modelagem atmosférica global
- ▶ Sistema de Equações Diferenciais Parciais
  - Eq. de conservação de momento
  - Eq. da continuidade
  - Eq. termodinâmica
  - Eq. da conservação da umidade
  - Outras (hidrostática, lei dos gases, ...)
- ▶ Forçantes do modelo (Física do modelo)
  - Radiação solar, formação de núvens, gases, ...

# Métodos de Resolução

## ▶ Métodos Numéricos

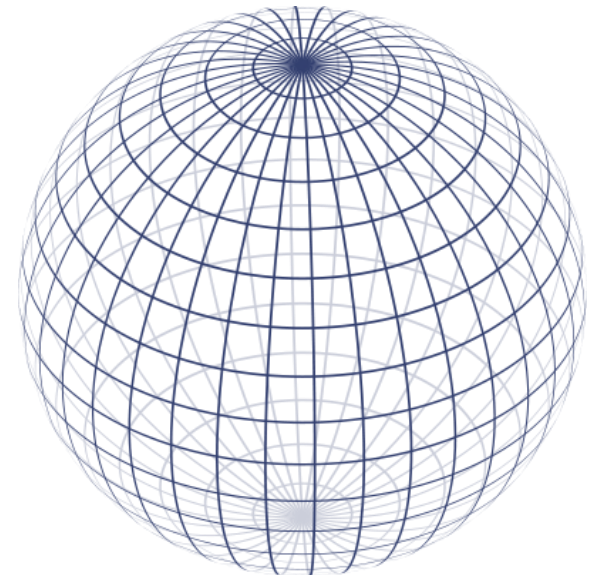
- Diferenças finitas
- Métodos Espectrais
- Volumes Finitos
- Elementos Finitos

## ▶ Malhas Numéricas

- Latitude – Longitude
- Gaussiana
- Icosaédrica
- Cubada

# Demanda Computacional

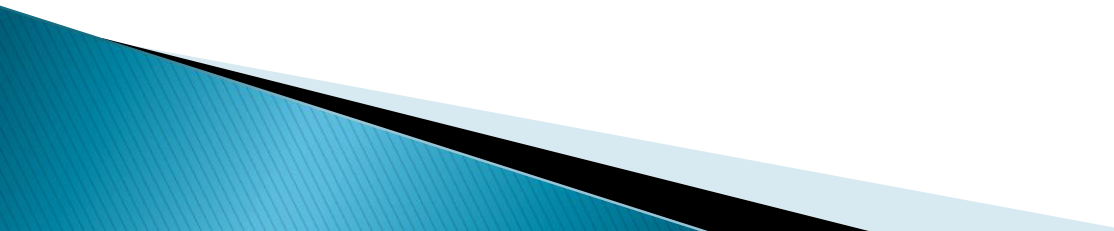
- ▶ Alta demanda computacional
  - Propriedades locais dos métodos numéricos torna o problema em geral altamente paralelizável
- ▶ Exemplo:
  - Malha latitude–longitude com resolução horizontal de aproximadamente 25 km e 90 camadas verticais
    - 1440 pontos de longitudes
    - 720 pontos de latitude
    - Total de aproximadamente **90 milhões de pontos** na malha tridimensional



# Primeira tentativa encontrada

- ▶ Em 2008 Michlakes e Vechharajani publicam: “GPU Acceleration of Numerical Weather Prediction”
- ▶ Teste com uma rotina da física do modelo WRF (Weather Research and Forecasting Model)
- ▶ NVIDIA 8800 GTX
- ▶ Tradução para CUDA manual (1500 linhas de código em F90)
- ▶ Ganhos de ~6x com comunicação, ~10x sem comunicação.

# Outras Tentativas

- ▶ Modelo CAM
    - Community Atmosphere Model
  - ▶ Modelo FIM
    - Flow-Following Icosahedral Model
  - ▶ Modelo NIM
    - Nonhydrostatic Icasahedral Model
  - ▶ Detalhes a seguir ...
- 

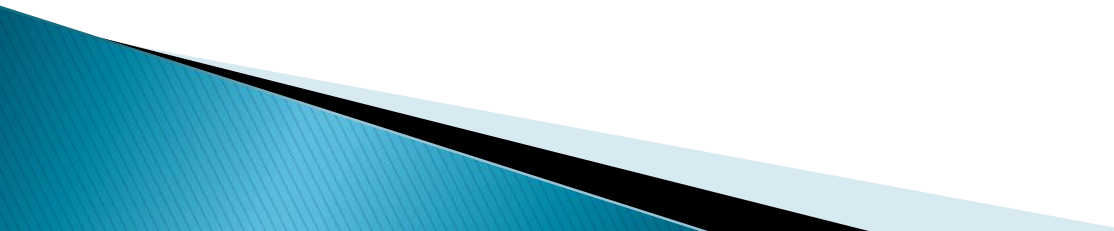
**PORTANDO MODELOS ATMOSFÉRICOS  
EXISTENTES PARA A GPU  
(O MODELO CAM)**



# Modelo CAM

- ▶ Community Atmosphere Model (CAM) – National Center for Atmospheric Research (NCAR) – Estados Unidos
- ▶ Artigo:
  - Rory Kelly (NCAR)
  - GPU Computing for Atmospheric Modeling
  - Computing in Science & Engineering – Aug. 2010
- ▶ Testes com um módulo pequeno do modelo e suas implicações no modelo completo

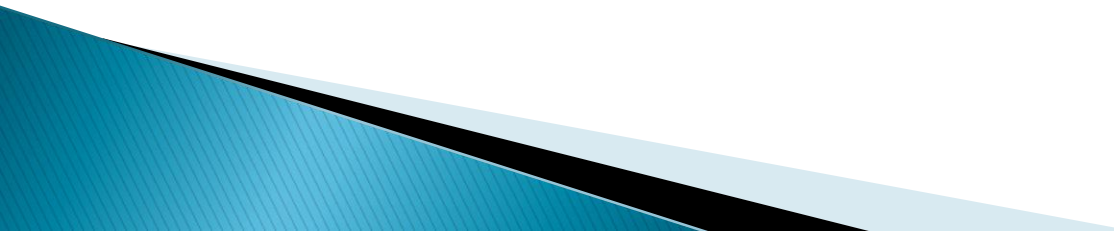
# O modelo CAM

- ▶ 139.000 linhas de código em Fortran 90
  - ▶ Latitude–longitude
  - ▶ 32.000 pontos na horizontal (colunas)
  - ▶ 26 níveis verticais
  
  - ▶ 3 núcleos principais: Euleriano, Semi–lagrangeano e de Volumes Finitos.
  - ▶ Física: Radiação, umidade, núvens, superfície, turbulência.
- 

# Radiação: RADCSWMX

- ▶ Dividir o espectro solar em intervalos
- ▶ Em cada intervalo calcular o espalhamento e absorção da luz na atmosphere, núvens, superfície e gases.
- ▶ Calcula a reflexão e refração da luz (RADDEDMX)
- ▶ É a rotina que mais consome tempo de computação
  
- ▶ RADDEDMX é a que mais consome tempo em RADCSWMX e é *embarrassing parallel* para cada coluna → GPU

# GPU

- ▶ NVIDIA GeForce 9800 GX2
  - ▶ 2 GPUs
  - ▶ Cada GPU com 16 SMP
  - ▶ Cada SMP com 8 cores, 2 unidades de funções especiais e 16Kb de memória local
  - ▶ Memória: 512MB DDR
  - ▶ PCIe (4Gb/seg)
  - ▶ CUDA
  - ▶ Precisão simples
- 

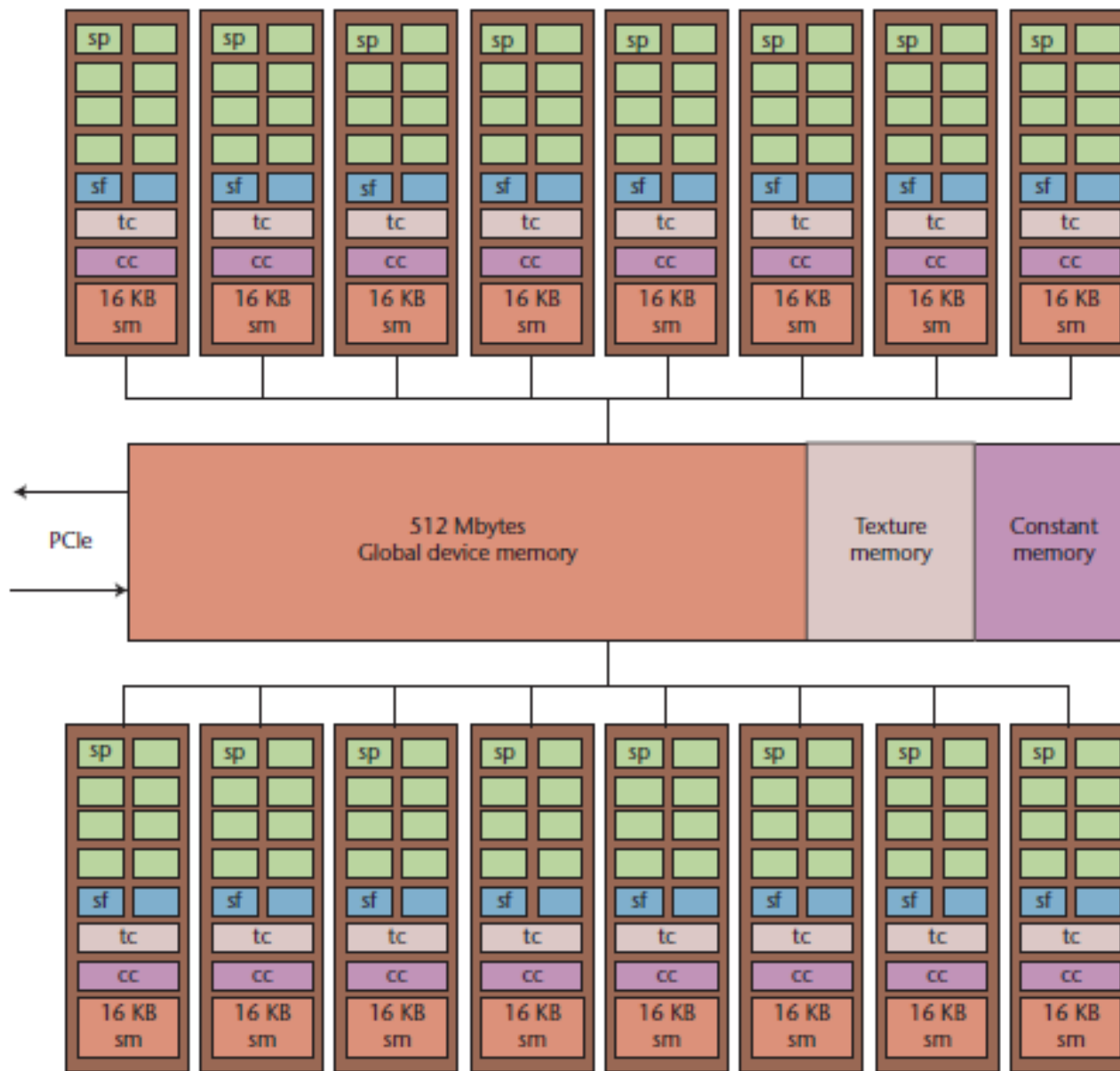


Figure 1. Major components of a G92 core on the GeForce 9800 GX2. The central portion shows global, texture, and constant memory. Each of the 16 boxes above and below corresponds to a streaming multiprocessor (SM). Within each SM, the eight green boxes are scalar processors, and the two blue boxes are special function units. The pink box is 16 Kbytes of local memory. The beige and purple boxes are caches for texture and constant memory, respectively.

# Performance

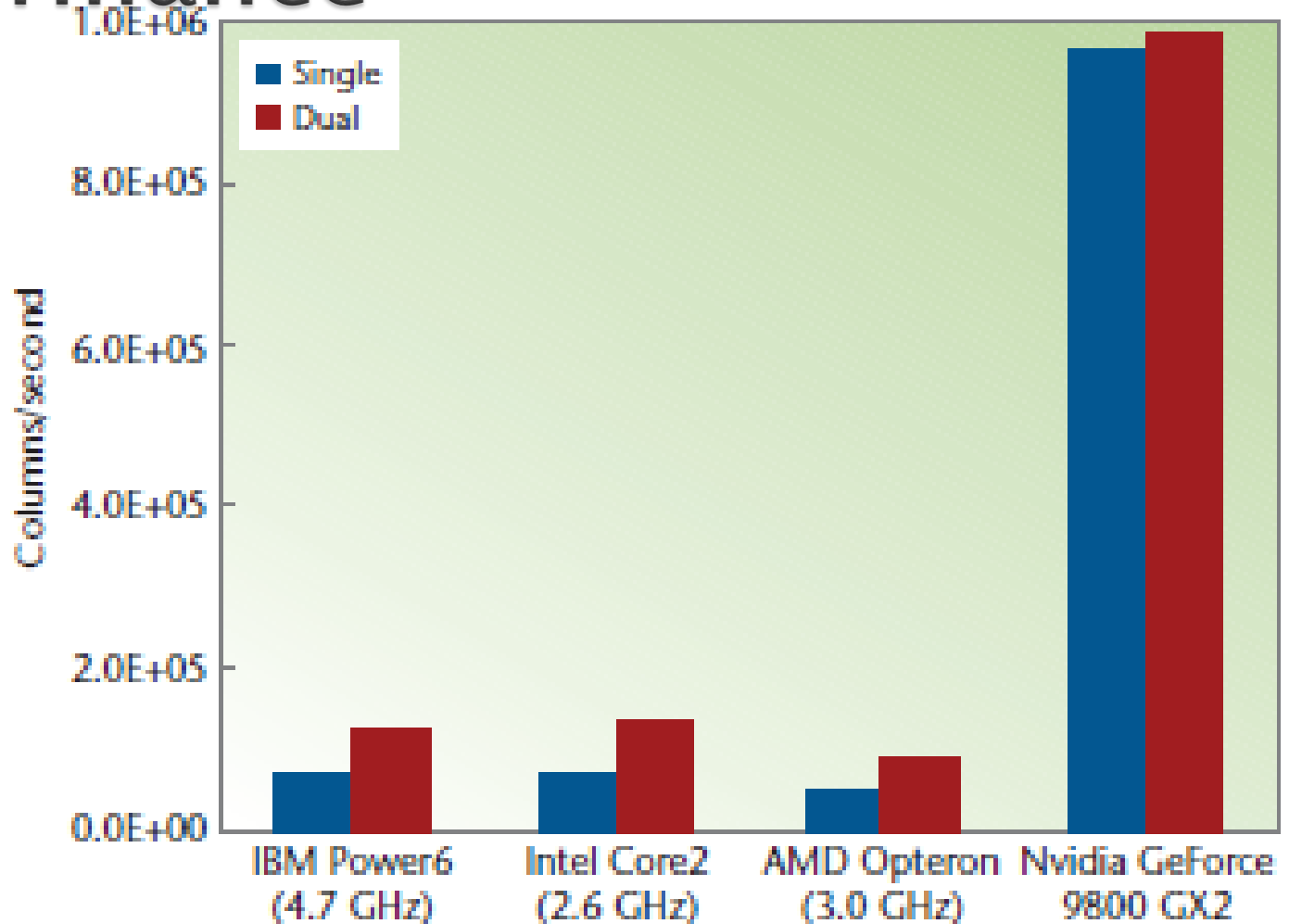


Figure 3. Comparing performance on one and two cores for traditional microprocessors and the Nvidia 9800 GX2 using one and both GPUs on the card. The GPU achieves a speedup of 14 to 20 times in the single-core case and a 7.5 to 11 times speedup in the dual-core case.

# Performance

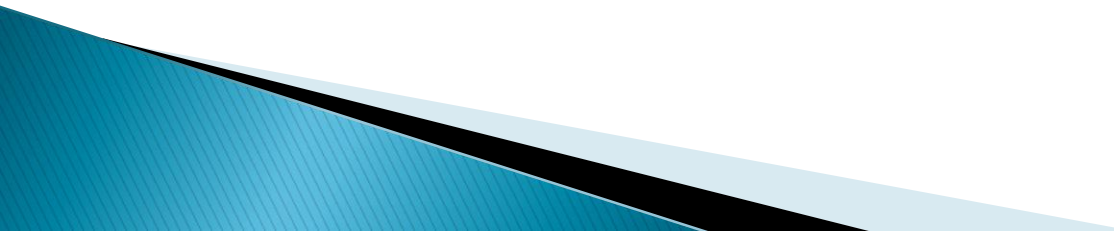
**Table 1. Comparing single core results for the Intel Core2 and the NVIDIA GPU, with and without direct-memory access (DMA) transfer time.**

	Intel Core2 2.6 GHz	Nvidia GPU (compute + DMA)	Nvidia GPU (compute only)
Time (ms)	462.64	33.08	2.02
Rate (Gflops/s)	0.782	10.93	179.09
% of peak	7.52	1.42	46.64

# Custo-Benefício

- ▶ RADCSWMX: 10.9 % de tempo de computação do modelo (Rotina mais cara)
- ▶ RADDEDMX: 3.6% de tempo de computação do modelo (3<sup>a</sup>. rotina mais cara de todo o código)
- ▶ 14 x Speed Up em RADDEDMX → 3% de ganho de performance no modelo como um todo

# Custo-Benefício

- ▶ Portando as 20 rotinas mais caras
  - ▶ Representam 48% do tempo de computação
  - ▶ 1000x de speed up
  
  - ▶ Modelo apenas chega perto de dobrar de performance
  - ▶ Alto custo de portabilidade para a reestruturação de muitas linhas de código.
- 

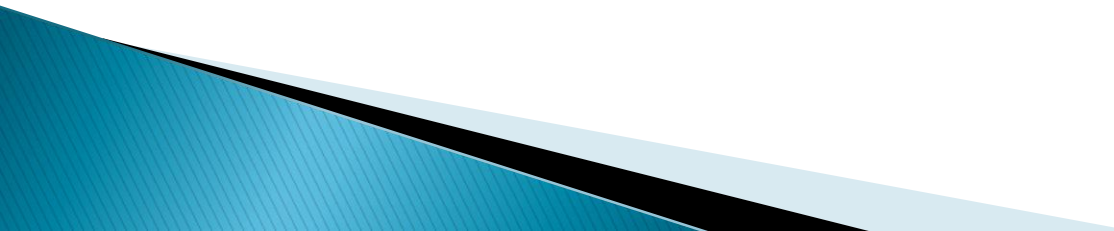
# Custo-Benefício

- ▶ Ganhos no modelo inteiro devem envolver reestruturação geral do código.

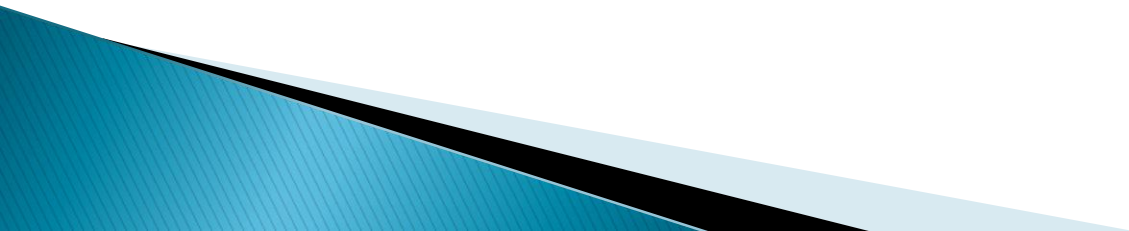
Mas...

- ▶ Nem todas as rotinas são facilmente paralelizáveis em GPU

# Conclusões Parciais

- ▶ O uso de GPU parece razoável apenas para experimentos de pequena escala, mas não em todo o modelo
  - ▶ Um futuro modelo rodando em GPU provavelmente será um modelo desenvolvido do zero para este fim.
  - ▶ No caso de múltiplas GPUs, cada uma ficaria apenas com uma porção pequena de colunas e o custo de comunicação se torna significativo.
- 

# AUTOPARALELISMO PARA GPU (MODELOS FIM E NIM )



# Fortran2CUDA

- ▶ Conversor automático de linguagem
- ▶ Desenvolvida no National Oceanic & Atmospheric Adm. (NOAA)
  
- ▶ Usa diretivas ACC
- ▶ EX:
  - ACC\$DO PARALLEL
  - ACC\$REGION
  
- ▶ Não traduz chamadas de Input/Output

# Exemplo – Fortran

```
subroutine loop_ij(its,nvl,.....)
```

```
...
```

```
!ACC$REGION (<nx>,<ny>) BEGIN
```

```
do ivl=1,nvl
```

```
!ACC$DO VECTOR
```

```
  do j=1,nx
```

```
!ACC$DO PARALLEL
```

```
  do i=1,ny
```

```
    do isn=1,nprox_ij(i,j)
```

```
      vnorm_ij(i,j,isn,ivl) =                &  
        sidevec_e_ij(i,j,2,isn)*ue_ij(i,j,isn,ivl) &  
        - sidevec_e_ij(i,j,1,isn)*ve_ij(i,j,isn,ivl)
```

```
    end do
```

```
  end do
```

```
end do
```

```
end do
```

```
!ACC$REGION END
```

# Exemplo – CUDA

```
#include <cutil.h>
#include "ftocmacros.h"

__global__ void loop_ij_Kernel1(int its,int nvl,int.....) {

    int i,j,ivl,isn;
    for (ivl=1;ivl<=nvl;ivl++) {
        j = threadIdx.x+1;
        i = blockIdx.x+1;
        for (isn=1;isn<=nprox_ij[FTNREF2D(i,j,nx,1,1)];isn++) {
            vnorm_ij[FTNREF4D(i,j,isn,ivl,nx,ny,npp,1,1,1,1)] =
                sidevec_e_ij[FTNREF4D(i,j,2,isn,nx,ny,nd,1,1,1,1)] *
                ue_ij[FTNREF4D(i,j,isn,ivl,nx,ny,npp,1,1,1,1)] -
                sidevec_e_ij[FTNREF4D(i,j,1,isn,nx,ny,nd,1,1,1,1)] *
                ve_ij[FTNREF4D(i,j,isn,ivl,nx,ny,npp,1,1,1,1)];
        }
    }
    return;
}
```

# Exemplo – CUDA (cont..)

```
extern "C" void loop_ij_ (int *its__G,int *nvl__G,int *npp__G,...) {  
  
    int its=*its__G;  
    ...  
    dim3 cuda_threads1(nx);  
    dim3 cuda_grids1(ny);  
  
    int *d_nprox_ij;  
    ...  
    cudaMalloc((void **) &d_nprox_ij,((nx)*(ny))*sizeof(int));  
    cudaMemcpy(d_nprox_ij,nprox_ij,((nx) (ny))*sizeof(int),  
               cudaMemcpyHostToDevice);  
  
    loop_ij_Kernel1 <<< cuda_grids1, cuda_threads1 >>>  
                    (its,nvl,npp,nd,nx,ny,d_nprox_ij,...);  
  
    cudaFree(d_nprox_ij); ...  
    cudaMemcpy(vnorm_ij,d_vnorm_ij,((nx)*(ny)*(npp)*(nvl))*  
               sizeof(float),cudaMemcpyDeviceToHost);  
}
```

# Considerações

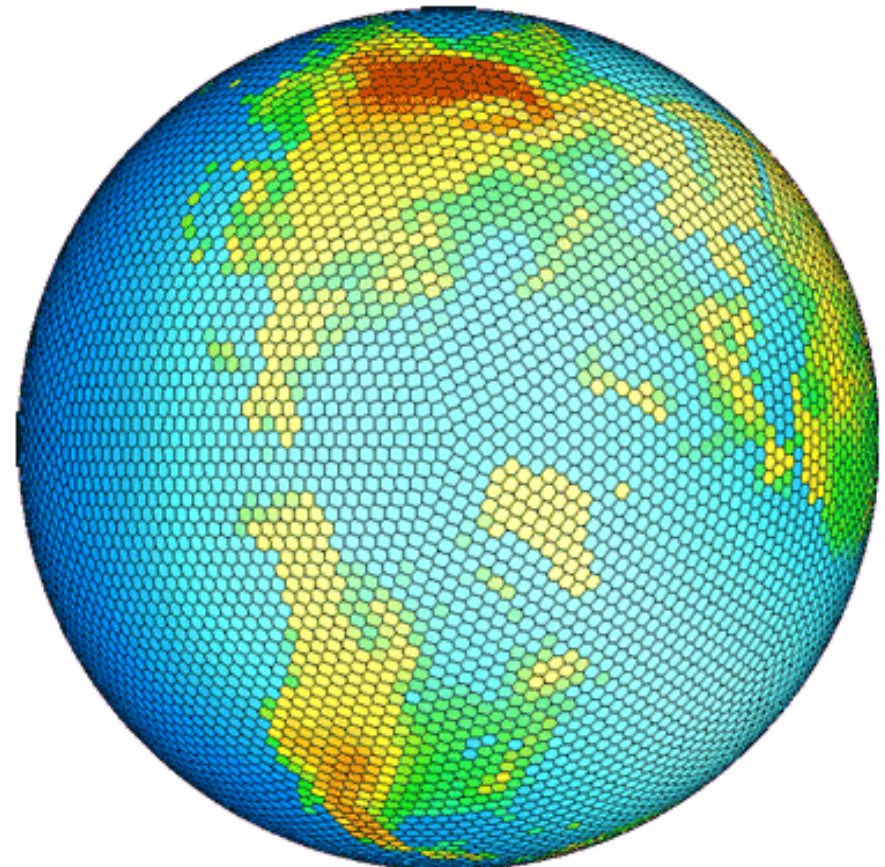
- ▶ Necessita de ajustes manuais
- ▶ É parte do conjunto de softwares de paralelismo automático SMS
  - SMS – The Scalable Modeling System
  - Paralelismo em MPI automático
- ▶ Existem outros pacotes auto-paralelizantes para GPU
  - PGI Accelerator: Fortran and C to CUDA C
  - CAPS HMPP C and Fortran to CUDA C
  - Goose: C to CUDA C Auto-parallelizing Compiler

# E isso funciona ?!

- ▶ Foi testado com os modelos FIM e NIM do NOAA
- ▶ Flow-Following Icosahedral Model (FIM)
- ▶ Nonhydrostatic Icasahedral Model (NIM)
- ▶ [www.esrl.noaa.gov/gsd/ab/ac/GPU\\_Parallelization\\_NIM.html](http://www.esrl.noaa.gov/gsd/ab/ac/GPU_Parallelization_NIM.html)

# Modelo FIM/NIM

- ▶ Volumes finitos em malha icosaédrica
- ▶ Resolve núvens
- ▶ Escala horizontal de 3 a 4 km



*The Icosahedral Grid used by the FIM and NIM models.*

# Testes FIM

- ▶ Rotinas que varrem todos os pontos de malha
  - 32 x 32 horizontais, 50 verticais
  - 1 pontos por thread

kernel	major features
timedif	two loops, no dependencies
getscl	six argument max & min function, indirect addressing
force	horizontal dependencies, many loops
load_ls	horizontal, vertical dependencies, many loops
fctprs	horizontal, vertical, inter-loop dependencies
fcttrc	Horizontal, inter-loop dependencies

# Testes FIM

## ▶ Timedif

- Tempos sem contar cópias:
  - CPU = 2ms / GPU = 0.110ms
- 5 dias para portar o código
- Esconde comunicação com computação na entrada, mas não na saída

## ▶ Getscl

- CPU = 1.8 ms / GPU = 0.78 ms

## ▶ Force

- CPU = 6.2 ms / GPU = 0.228 ms

## ▶ ...

# Testes NIM

- ▶ Paralelismo com apenas um nó:

**“The CUDA code runs 25 times faster than on the CPU (Intel Harpertown)”**

- Toda dinâmica na GPU (reduz tranf. dados)
  - Número de níveis verticais (96) múltiplo de Warps
  - Quebra de rotinas grandes em pequenos kernels
- ▶ Paralelismo com vários nós:
    - Vai usar SMS. A fazer (comentário de Outubro de 2010 no site)

# NIM - 2652 pontos

	Harpertown CPU Time	F2C-ACC CUDA-C Tesla GPU Time	HMPP Tesla GPU Time	PGI Tesla GPU Time	F2C-ACC CUDA-C Fermi GPU Time
vdmints	88.86	2.05	2.35	4.78	1.92
vdmintv	37.73	0.94	0.98	0.97	0.75
flux	17.97	0.55	1.05	2.51	0.30
vdn	12.77	0.56	0.73	--	0.53
diag	5.13	0.086	0.085	0.077	0.09
force	5.34	0.11	0.19		0.08
trisol	8.41	1.38	<b>1.38</b>	--	1.14

# CONCLUSÕES

# Custos

## DOE Jaguar System

- 2.3 PetaFlops
- 250,000 CPUs
- 284 cabinets
- 7-10 MW power
- ~ \$100 million
- **Reliability in hours**



## Equivalent GPU System

- 2.3 PetaFlop
- 2000 Fermi GPUs
- 20 cabinets
- 1.0 MW power
- ~ \$10 million
- **Reliability in weeks**

- ▶ Modelos com alta resolução (que resolvem núvens) precisam de computadores com petaflops
- ▶ Inviabilidade de sistemas com muitas CPU (> 100 mil) : Energia, resfriamento, custo, escalabilidade



Valmont  
Power Plant  
~200 MegaWatts  
Boulder, CO

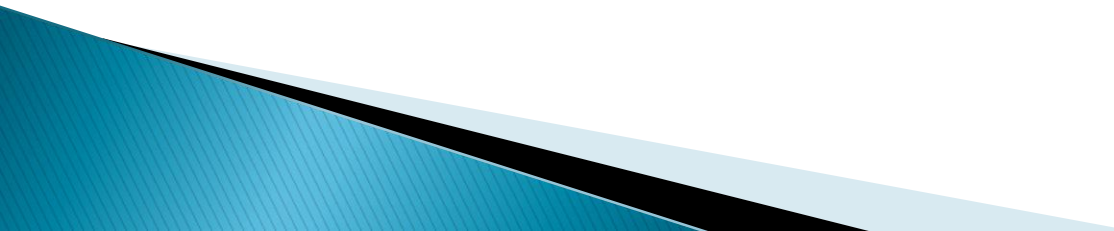
Fonte: Apresentação de Mark Govett - Nov. 2010

# Memória

- ▶ A memória global de GPU limita o grau de escalonamento

	G4	G5	G6	G7	G8	G9	G10	G11
<b>resolution</b>	480KM	240KM	120KM	60KM	30 KM	15 KM	7 KM	3.5 KM
<b>horizontal points</b>	2.5K	10K	40K	160K	640K	2560K	10,000K	40,000K
<b>memory</b>	.25GB	1GB	4GB	16GB	64GB			
<b># GPUs</b>	1	1	1	4	16	64	256	1024

# Conclusões Gerais

- ▶ Alto custo de recodificação para GPU para um ganho não tão grande
  - ▶ Autoparalelismo?
  - ▶ Uso mais indicado para pequenos módulos e não paralelismo global de modelos
  - ▶ Novos modelos já codificados para GPU ?
- 

**“CLUSTERS AREN’T GOING AWAY,  
THEY ARE JUST BECOMING MORE  
COMPLICATED”**

**RORY KELLY**

