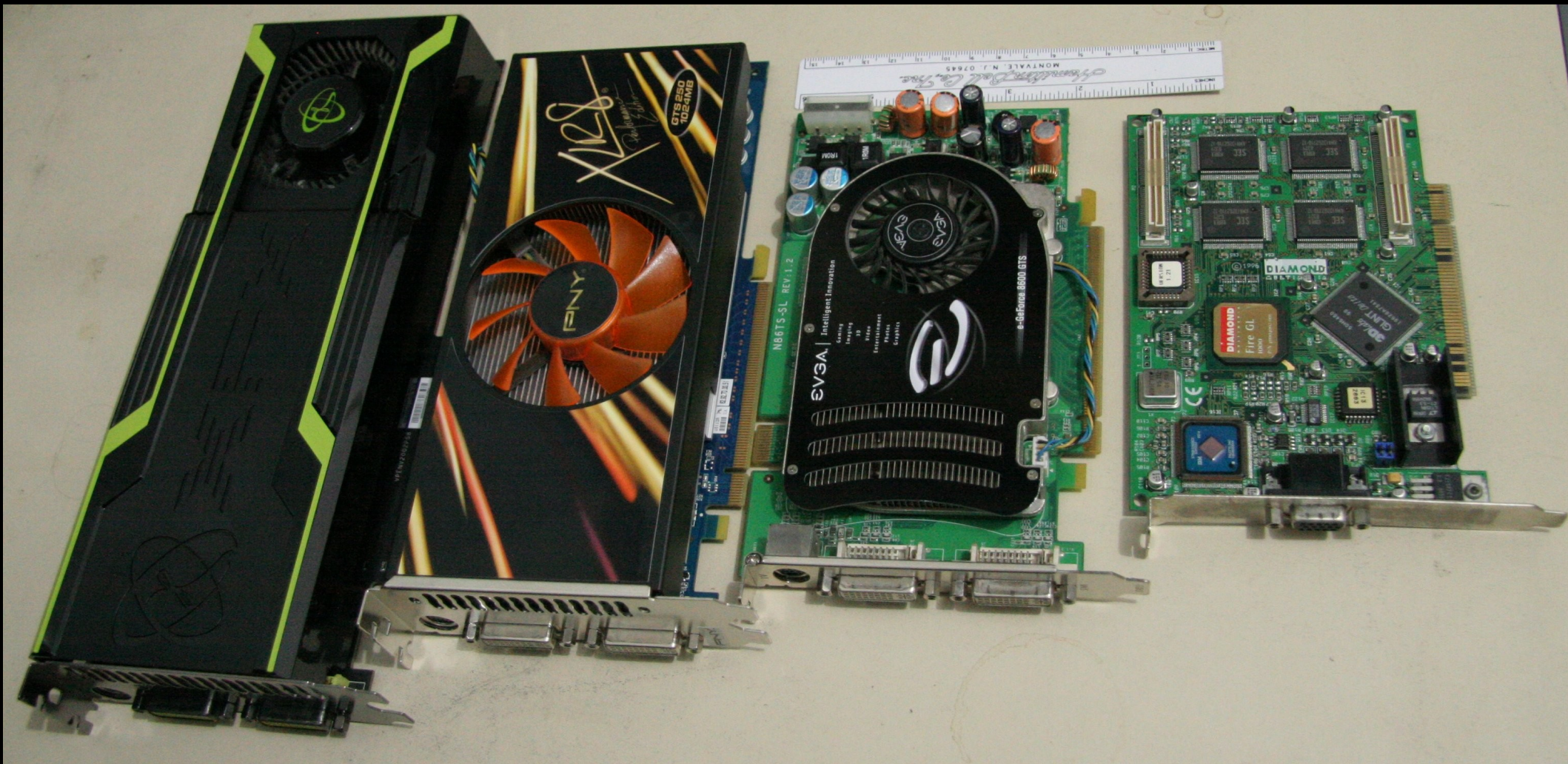


Computação com GPUs em problemas astronômicos



Nvidia GeForce
GTX 260 Core 216
(2008)
216 núcleos 875 Gflops

Nvidia GeForce
GTS 250
(2009)
128 núcleos 705 Gflops

Nvidia GeForce
8600 GTS
(2007)
32 núcleos 139 Gflops

Diamond FireGL
1000
(1996)
?

Panorama

GPUs – por que usar?

Histórico

GPUs x CPUs

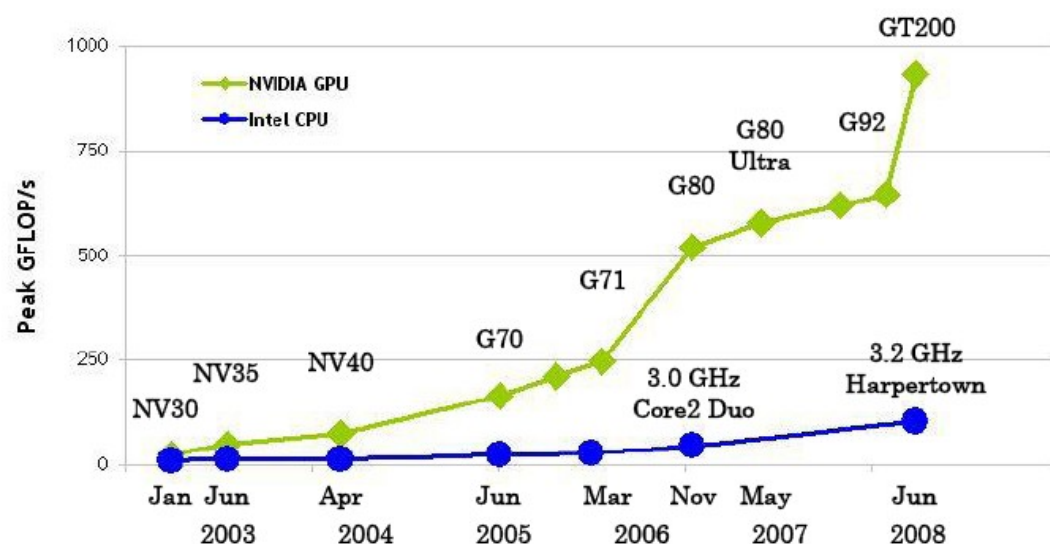
Plataformas disponíveis

Exemplos de uso

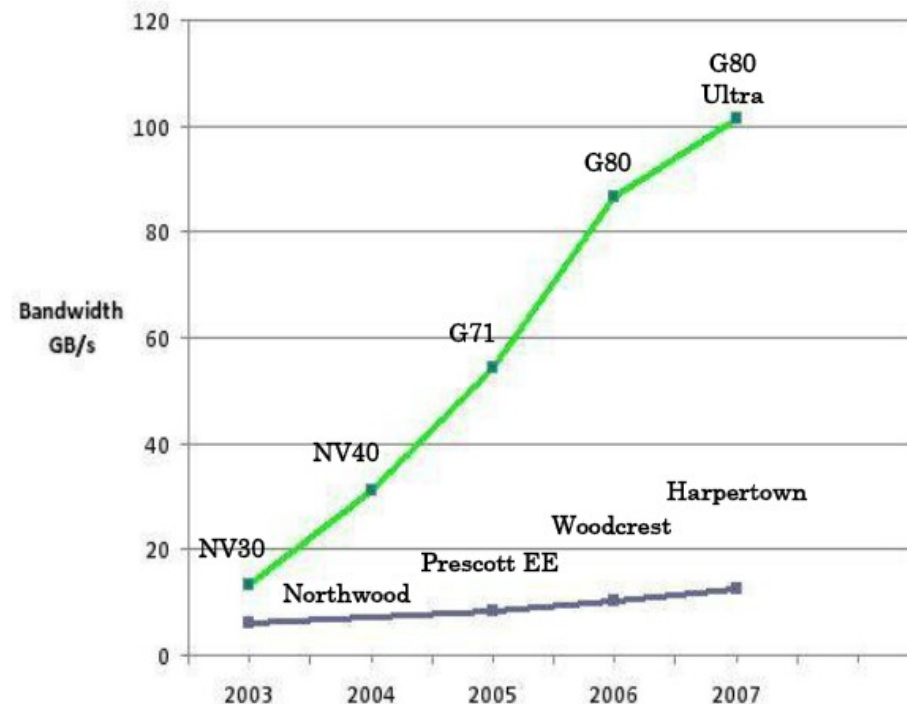
Exemplos de aplicações

GPUs no INCT-A

Trabalho em desenvolvimento



GT200 = GeForce GTX 280	G71 = GeForce 7900 GTX	NV35 = GeForce FX 5950 Ultra
G92 = GeForce 9800 GTX	G70 = GeForce 7800 GTX	NV30 = GeForce FX 5800
G80 = GeForce 8800 GTX	NV40 = GeForce 6800 Ultra	



GPUs – por que usar?

Graphical Processing Units

Criadas para livrar a CPU de processar gráficos em 2D e 3D e vídeos.

Trabalho simples, mas pesado (grande volume de dados, muitos quadros por segundo).

Complexidade exigida para cenas em 3D levou ao desenvolvimento de shaders programáveis – capazes de executar código arbitrário.

Hoje executam operações 1-2 ordens de magnitude mais rápido que CPUs de preços semelhantes.

Histórico

Até ~1990, todo o trabalho era feito pela CPU.

Interfaces gráficas exigiam muitas operações em imagens (2D), ocupando muito a CPU.

~1990-1995, aceleradores gráficos criados para realizar operações em imagens.

1992 – OpenGL (Open **G**raphics **L**ibrary), criado pela SGI.
Padrão aberto para API (Application **P**rogramming **I**nterface) para gerar cenas em 3D.

1997 – OpenGL implementado em GPUs, livrando as CPUs deste trabalho pesado.

MS tenta competir com Direct3D (proprietário, só Windows).

Histórico - O que faziam as GPUs?

Cenas são definidas por objetos que emitem, absorvem, refletem, espalham, refratam luz, e uma câmera.

O programador só cria os objetos, o trabalho de calcular o resultado é feito pelo OpenGL.

As mesmas operações são feitas em um volume grande de elementos em arrays: vértices (dos objetos) ou pixels (das imagens).

Hardware otimizado para operações vetoriais (com arrays) em memória própria (mais rápida).

Cada núcleo é muito mais simples que o de uma CPU, só faz as poucas operações necessárias.

Histórico - O que fazem as GPUs?

Desenvolvimento movido para computação gráfica (principalmente jogos 3D).

Maior demanda levou a maior performance:

Placas com muitos núcleos (hoje, centenas) e memória própria mais rápida para operações em arrays.

Maior complexidade levou a shaders programáveis:

Podem executar programas arbitrários (kernels), os mesmos para cada vertex ou pixel, para determinar suas propriedades.

Abrem a possibilidade de kernels para outros usos.

Modelos recentes baseados em GPUs gráficas, alteradas para uso apenas computacional.

GPUs x CPUs

Ambas têm seguido aproximadamente a lei de Moore

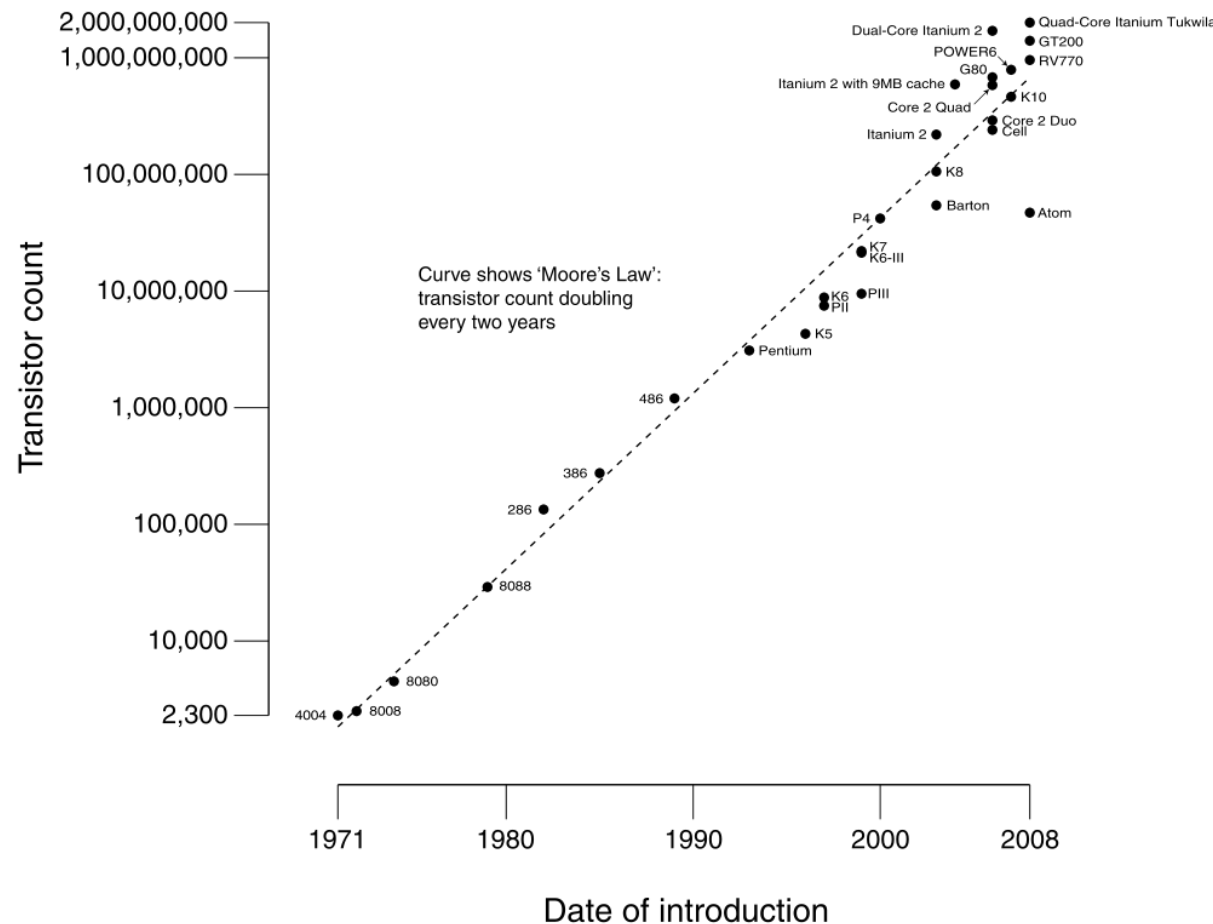
Até 2005 (Pentium 4) o aumento de rapidez dos programas era automático: CPUs mais rápidas.

Com o teto em ~3GHz, CPUs tiveram que ser redesenhadas para usar mais núcleos.

Software também precisou ser mudado para usar paralelismo.

GPUs sempre tiveram uma tarefa paralela, evoluíram continuamente.

CPU Transistor Counts 1971-2008 & Moore's Law



GPUs x CPUs

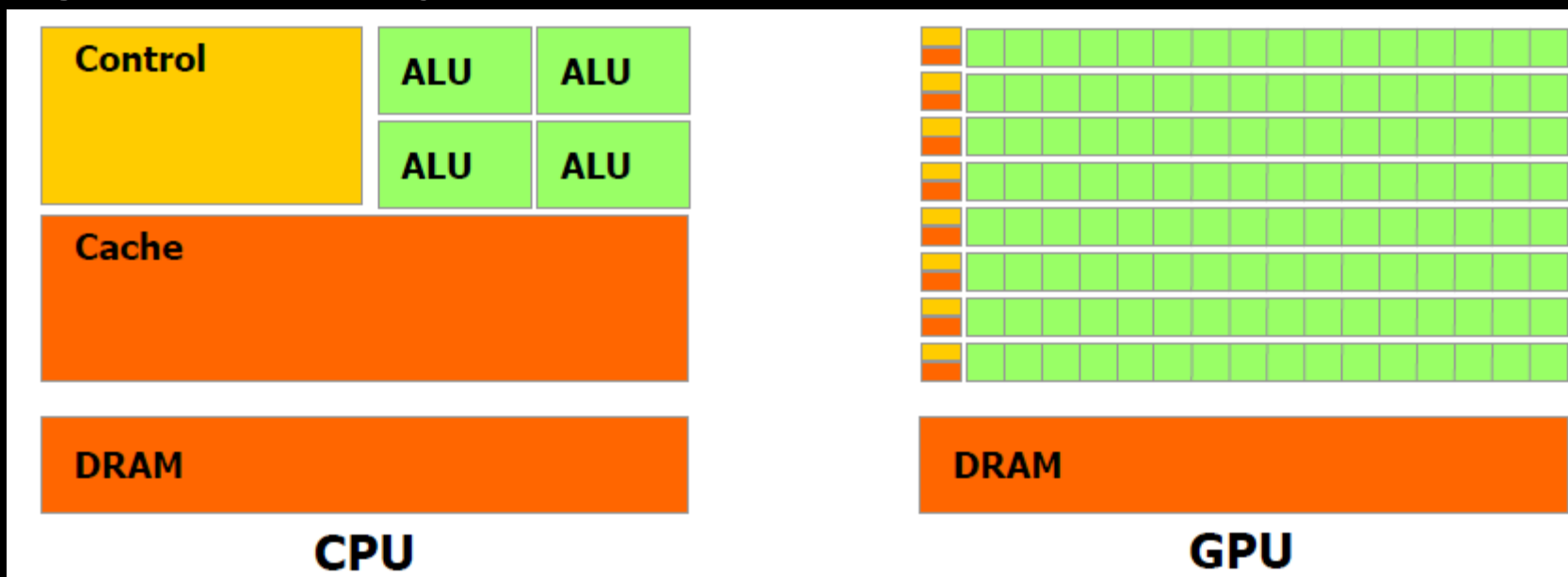
Nvidia GeForce GTX 480 (Fermi):

3×10^9 transistores, 480 núcleos, ~ 1 Tflops, $\sim \$500$

Intel Core i7 980x:

1.2×10^9 transistores, 6 núcleos (HT), ~ 0.1 Tflops, $\sim \$1000$

CPUs têm núcleos mais complexos e mais cache: melhores para operações independentes, acesso aleatório à memória e operações com poucos dados. GPUs melhores para tarefas simples em grandes arrays.



GPUs x CPUs

Não é mais viável só esperar as CPUs ficarem mais rápidas para programas tradicionais (seriais).

Programas têm que ser paralelizados, mesmo que só para CPUs.

Os problemas com repetição de operações em arrays (paralelismo de dados) são os mais apropriados para GPUs, que são SIMD (Single Instruction, Multiple Data).

Problemas de paralelismo de tarefas são mais apropriados para CPUs de muitos núcleos ou clusters de várias CPUs, através de OpenMP (SIMD) ou MPI (MIMD/SIMD).

Não é trivial fazer o melhor uso destes recursos. Diferentes sistemas têm diferentes vantagens e diferentes gargalos.

Plataformas disponíveis

Toda a programação para GPUs usava OpenGL:

Necessário expressar tudo como operações gráficas (ex. operações sobre texturas): difícil, baixo nível e desajeitado.

Apenas precisão simples.

APIs específicas para GPGPU (General Purpose Computing on GPUs):

CUDA (2007) - Compute Unified Device Architecture

Desenvolvida pela Nvidia, juntamente com o hardware.

Proprietária mas gratuita.

A mais madura e amigável hoje.

OpenCL (2008) – Open Computation Language

Desenvolvida pelo Khronos Group.

Aberta, idéia semelhante a OpenGL.

A mais flexível.

Direct Compute (2009) - Baseada no DirectX 11

A mais primitiva e limitada, a menos usada.

CUDA

Desenvolvida desde 2007, em conjunto com o hardware Nvidia.

Precisão simples e dupla (alguns modelos), e operações com inteiros e bitwise.

Possui memória compartilhada entre os ramos.

Otimizada para o hardware.

Usada pelas GPUs GeForce (consumidor), Quadro (workstations), Tesla (específicas para computação, não são placas de vídeo, usadas em clusters).

Rápida quando usando a própria memória – gargalo fica na transferência entre CPU e GPU.

Tem a maior variedade de bibliotecas e APIs.

CUDA – bibliotecas e APIs

CUDA SDK – Nvidia (gratuita):

Compilador (Open64), debugger e profiler para kernels (C/C++)

Suporte a OpenCL

Bibliotecas BLAS e FFT (C, com interfaces para Fortran).

CUDA NPP – Nvidia (gratuita):

Biblioteca para processamento de imagens (C).

PyCUDA (gratuita)

Interface aberta para escrever kernels com Python+Numpy.

Única linguagem de alto nível para kernels.

Jacket - Acclereyes

Interface e bibliotecas para MATLAB.

JCUDA, CUDA.NET – Hoopoe (gratuitas)

Interface e bibliotecas para Java e .NET.

Jcuda, Jcublas, Jcufft (gratuitas)

Interfaces abertas para CUDA e bibliotecas para Java.

CUDA – bibliotecas e APIs

CULAtools – EM Photonics

~LAPACK híbrido, C, C++.

CUDA Fortran PGI

Compilador para kernels escritos em Fortran, C, C++.

GPULib – Tech X Corporation (gratuita para uso acadêmico)

Interface e biblioteca para operações com arrays (IDL, MATLAB).

Fortran CUDA – Hoopoe (gratuito, mas abandonado)

Compilador para kernels em Fortran.

Biblioteca FFT.

HMPP Workbench – CAPS

Compilador para programas híbridos C, Fortran

Opera com CUDA, OpenCL, e CPUs de múltiplos núcleos.

Através de compiladores Intel, gcc, gfortran, PGI, Sun, Open64.

CUDA – bibliotecas e APIs

MAGMA – Univ. Tennessee, et al. (gratuita)

Biblioteca híbrida (CPU+GPU) BLAS e LAPACK (C).

Ecolib – GPU Computing

BLAS, LAPACK, números aleatórios e VaR (C++).

R+GPU (gratuita)

Funções de R reimplementadas usando CUDA.

Libra – GPU Systems

Compilador híbrido (C++), para OpenCL, OpenGL, CUDA, CPUs.

Flagon

API, CUFFT, CUBLAS abertas para Fortran.

F2C-ACC – NOAA

Tradutor aberto de Fortran para C+CUDA (ainda não completo).

CUDA – bibliotecas e APIs

Nvidia Parallel Insight (beta gratuito)
IDE para Visual Studio.

Thrust
Biblioteca aberta para funcionalidade semelhante à C++ STL.

OpenCL

Padrão aberto, desenvolvido pelo Khronos Group.

API independente de hardware – pode ser implementada por CPUs, clusters, GPUs, OpenGL, CUDA.

Programas com OpenCL podem ser executados em qualquer plataforma OpenCL, independente do hardware.

Deliberadamente de baixo nível (OpenCL C, próximo ao hardware, apesar de abstrato).

Ainda não tão evoluído como CUDA, há poucas implementações, bibliotecas e interfaces para mais alto nível.

Operações SIMD semelhantes a OpenMP, agrupando os dados em workgroups. Separa a memória em privada (elemento), local (workgroup), global e do host.

Alguns dispositivos (CPUs) também têm paralelismo de tarefas.

OpenCL - implementações

OpenCL C (padrão aberto)

Linguagem para programar os kernels.

Subconjunto do C, adicionado de funções para transporte de dados entre memórias e controle dos ramos.

Suportado em todas as GPUs Nvidia com CUDA.

Suportado pelas GPUs com ATI Stream.

HMPP Workbench – CAPS

Compilador para programas híbridos em C, Fortran

Opera com CUDA, OpenCL, e CPUs de múltiplos núcleos.

Através de compiladores Intel, gcc, gfortran, PGI, Sun, Open64.

Libra – GPU Systems

Compilador híbrido (C++), para OpenCL, OpenGL, CUDA, CPUs.

Interfaces para Python (Clyther, PyOpenCL), Ruby, .NET.

ViennaCL

Implementação aberta de BLAS, interface para C++ e MATLAB.

Exemplos de uso – alto nível (CUDA)

GPU: Nvidia GeForce GTX 260 Core 216 – 216 núcleos, 0.9Tflops

CPU: Intel Core i7 920, 4 núcleos, 2.7GHz

Ambos ~\$250 em outubro/2009

$\ln(\Gamma(z))$, calculada com um kernel específico (GPULib, IDL), 11 vezes para 10^6 elementos:

Apenas IDL:

```
for i = 0, niter do er = lngamma(x)
```

GPU:

```
; create gpu variables.
```

```
x_gpu = gpuFltArr(nx)
```

```
res_gpu = gpuFltarr(nx)
```

```
gpuPutArr, x, x_gpu
```

```
for i=0, niter do gpuLGamma, x_gpu, res_gpu
```

```
gpuGetArr,res_gpu, x
```

~45x mais rápido em precisão simples, ~22x em precisão dupla.

Exemplos de uso – alto nível (CUDA)

FFTs, 3x3000x3000

Apenas IDL:

```
fimg_clean = fltarr(3, nx, ny)
for c = 0, 2 do begin
    fimg_fft = fft(fimg[c, *, *])
    fimg_decon = fimg_fft / psf_fft
    fimg_clean[c, *, *] = float(fft(fimg_decon, -1))
End
```

GPU:

```
fimg_clean = fltarr(3, nx, ny)
for c = 0, 2 do begin
    gpuFFT, reform(fimg[c, *, *]), gpu_fimg, /DIM2D
    gpuDiv, gpu_fimg, gpu_psf_fft, gpu_fimg
    gpuFFt, gpu_fimg, gpu_fimg, /DIM2D, /INVERSE
    gpuFloat, gpu_fimg, gpu_fimg_clean_fix
    gpuGetArr, gpu_fimg_clean_fix, fimg_clean_gpu
    fimg_clean[c, *, *] = fimg_clean_gpu
End
```

9x mais rápido.

Interpolação 2D, 2048x2048: 22 vezes mais rápido

Exemplos de uso – alto nível (CUDA)

GPULib x IDL (por Michael Galloy):

CPU calculations performed on a 2.40GHz Core 2 Duo

GPU calculations performed on a NVIDIA Tesla C1060

10^3 iterations with 10^6 element arrays

CPU calculation: 144.8 secs

Procedure forms: 7.3 secs

Function forms: 15.4 secs

Function forms with LHS: 7.2 secs

Operator forms: 15.4 secs

Custom kernel: ~1.5secs

Exemplos de uso – baixo nível (CUDA)

Kernel para adição de dois vetores, $C=A+B$:

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}
```

ThreadIdx identifica cada ramo, usado aqui para separar que elementos cada um processa. Pode ser 1D, 2D ou 3D.

Ramos podem ser agrupados em blocos, 1D ou 2D.

Programa que usa o kernel:

```
int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```

Exemplos de uso – baixo nível (CUDA)

Kernel para multiplicação de matrizes:

```
attributes(global) subroutine MMUL_KERNEL( A,B,C,N,M,L)
!
real,device :: A(N,M),B(M,L),C(N,L)
integer,value :: N,M,L
integer :: i,j,kb,k,tx,ty
real,shared :: Ab(16,16), Bb(16,16)
real :: Cij
!
tx = threadidx%x ; ty = threadidx%y
i = (blockidx%x-1) * 16 + tx
j = (blockidx%y-1) * 16 + ty
Cij = 0.0
do kb = 1, M, 16
  ! Fetch one element each into Ab and Bb; note that 16x16 = 256
  ! threads in this thread-block are fetching separate elements
  ! of Ab and Bb
  Ab(tx,ty) = A(i,kb+ty-1)
  Bb(tx,ty) = B(kb+tx-1,j)
  ! Wait until all elements of Ab and Bb are filled
  call syncthreads()
  do k = 1, 16
    Cij = Cij + Ab(tx,k) * Bb(k,ty)
  enddo
  ! Wait until all threads in the thread-block finish with
  ! this iteration's Ab and Bb
  call syncthreads()
enddo
C(i,j) = Cij
!
end subroutine
```

Exemplos de uso – baixo nível (CUDA)

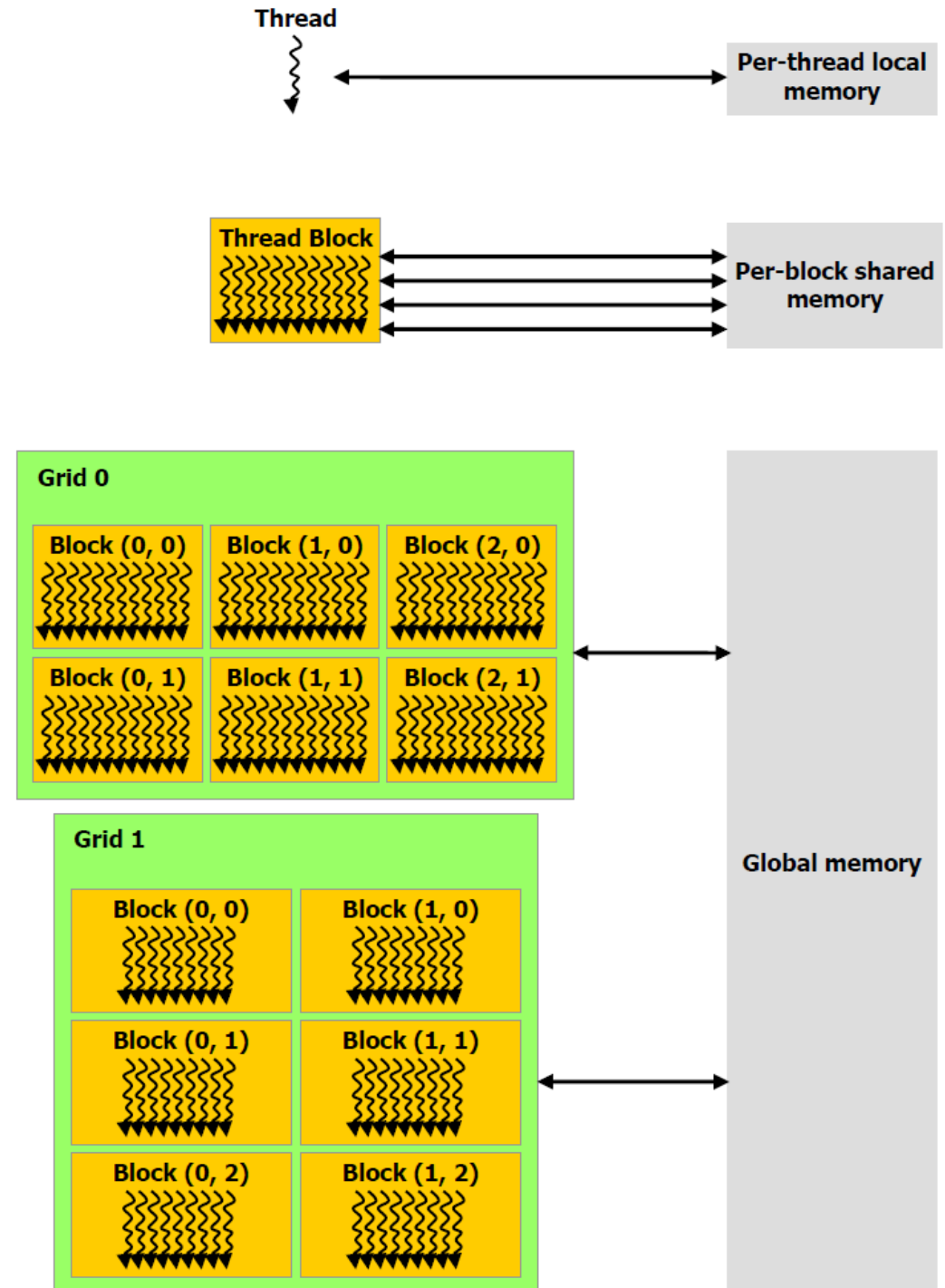
Programa que usa o kernel:

```
subroutine mmul( A, B, C )
!
  use cudafor
  real, dimension(:, :) :: A, B, C
  integer :: N, M, L
  real, device, allocatable, dimension(:, :) :: Adev, Bdev, Cdev
  type(dim3) :: dimGrid, dimBlock
!
  N = size(A,1) ; M = size(A,2) ; L = size(B,2)
  allocate( Adev(N,M), Bdev(M,L), Cdev(N,L) )
  Adev = A(1:N,1:M)
  Bdev = B(1:M,1:L)
  dimGrid = dim3( N/16, L/16, 1 )
  dimBlock = dim3( 16, 16, 1 )
  call mmul_kernel<<<dimGrid,dimBlock>>>( Adev, Bdev, Cdev, N, M, L )
  C(1:N,1:M) = Cdev
  deallocate( Adev, Bdev, Cdev )
!
end subroutine
```

Exemplos de uso baixo nível (CUDA)

Organização das variáveis e
divisão da execução em
ramos e blocos, de acordo o
que se encaixa melhor no
algoritmo.

Para eficiência, número de
ramos por bloco deve ser
múltiplo de 32 (e deve ser
menor que o limite máximo).



Exemplos de uso baixo nível (CUDA)

Partes seriais executadas na
CPU (host), partes paralelas
por kernels na GPU (device).

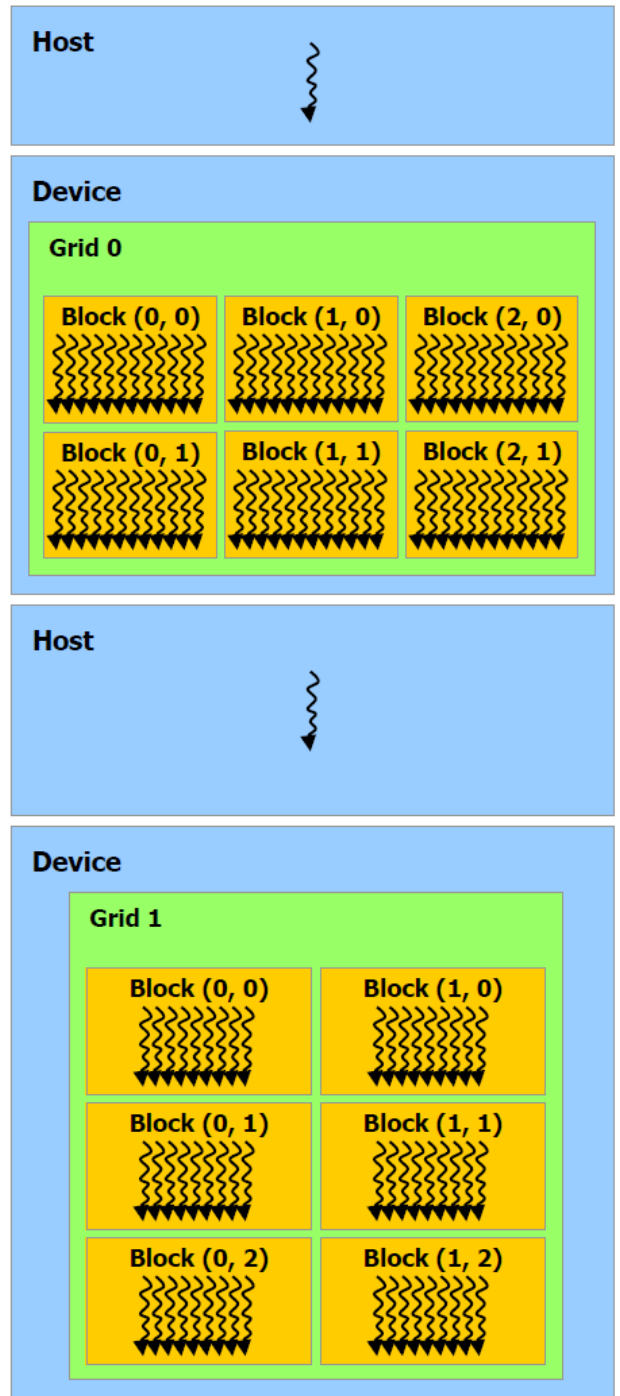
C Program Sequential Execution

Serial code

Parallel kernel
Kernel0<<<<>>>>()

Serial code

Parallel kernel
Kernel1<<<<>>>>()



Exemplos de aplicações astronômicas

Artigos com GPU/CUDA/OpenCL no título no ADS:

2010 (jan-ago) - 29

2009 - 14

2008 - 9

2007 - 1

CFD, MHD, N corpos, relatividade, PCA, processamento de dados.

Belleman et al., 2008: High performance direct gravitational N-body simulations on graphics processing units II: An implementation in CUDA

Ainda da época (2006-2007) de precisão simples, e software rudimentar.

GeForce 8800GTX (128 núcleos, 0.5Tflops), comparável a um GRAPE-6Af (4 chips com cálculo da força em hardware, 0.12 Gflops).

Para $N > 10^5$, ~100 vezes mais rápido que 2 Intel Xeon 3.4GHz.

Exemplos de aplicações astronômicas

Wong et al., 2009: Efficient magnetohydrodynamic simulations on graphics processing units with CUDA

GeForce GTX 295 (2x 240 núcleos, 1.8Tflops), comparada a Intel Core 2 Quad Q9650 3GHz:

Tempo CPU/GPU em precisão simples:

1D, 4096 pontos : 10

2D, 1024^2 pontos: 200

3D, 128^3 pontos: 84

Precisão dupla com ~60% da performance de precisão simples.

GPUs no INCT-A

Um SGI Altix XE 1300 Cluster, para uso nacional:

- 2 Compute nodes C1103 cada um com:

 - 2 Intel Xeon 5650 2.66-GHz (6 núcleos)

 - 48GB RAM DDR3 1333MHz RDIMM

 - 2 NVidia Tesla C2050, 448 núcleos, ~1Tflops para precisão simples.

6 GeForce GTX 280, para uso local nas instituições para desenvolvimento do software.

Workshop na Unicsul / cursos pelo LNCC (?)

Trabalho em desenvolvimento

Atual:

Transferência radiativa por ordenadas discretas:

Manipulação de grandes matrizes, cálculo de autovalores e autovetores.

Implementação por CULAtools em C++, acessível por DLM em IDL.

Futuro:

Transferência radiativa por ray-tracing (Monte-Carlo)

Mesma operação repetida para grandes números de fótons para cada comprimento de onda.

O mesmo problema físico para o qual GPUs foram desenvolvidas.

Em ambos casos, repetições para dezenas de comprimentos de onda para cada espectro, para milhões de espectros.