Practicals : Hands on Session for Programming GPUs

Paul Mullowney
paulm@txcorp.com

Tech-X Corporation
5621 Arapahoe Ave., Boulder, CO 80303
http://www.txcorp.com

Tech-X UK Ltd
Daresbury Innovation Centre
Keckwick Lane
Daresbury
Cheshire WA4 4FS, UK
http://www.txcorp.co.uk

Tech-X GmbH
Claridenstrasse 25
CH-8027 Zurich
Switzerland
http://www.txcorp.ch

TECH-X CORPORATION

Outline

- **A Few Examples to Get you Started**

- A Real Problem : A Wave Equation Solver on the GPU
  - Problem Description
    - Mathematical Algorithm
    - Example Code Structure
  - GPU Implementation using PGI Cuda Fortran
    - Simplistic Approach
    - A more advanced implementation ... Aligned memory
    - Other tricks?
    - Strategies for minimizing then cost of host-device data transfer
    - Analysis
  - Strategies for Porting Codes to GPUs

Getting/Building the first CUDA example

- **On the machine, do**

```
source /opt/env/pgi.env
```

- **Copy file from /opt/training/gpu to your home**

```
cp /home/quest/Software_depot/training/gpu/first.cuf .
```

- **Compile**

```
pgf90 first.cuf
```

- **Run**

```
./a.out
```

Running/Modifying the first CUDA example

- **GOAL: edit `first.cuf` to get some non-zero result that runs on a GPU.**

  Write a kernel that computes the sqrt() of all elements of a "small" vector element-wise.

- **With your favourite editor, open `first.cuf` and add the following line after the last variable declaration.**

If you sit on the left side of the room, do

```
ierr = cudaSetDevice(0)
```

If you sit on the right side of the room, do

```
ierr = cudaSetDevice(1)
```

Running/Modifying the first CUDA example

Let's learn something about the device we're running on. Add the followind variable declaration.

```
type(cudadeviceprop) :: prop
```

For the left side of the room, add the code below cudaSetDevice()

```
ierr = cudaGetDeviceProperties(prop, 0)
print *, "Name of my GPU is ", prop%name
```

For the right side of the room, do

```
ierr = cudaGetDeviceProperties(prop, 1)
print *, "Name of my GPU is ", prop%name
```

Hre's a possibly complete list of the properties:

```
name, totalGlobalMem, regsPerBlock, warpSize,
  memPitch, maxThreadsPerBlock, maxThreadsDim(3),
  maxGridSize(3), clockRate, totalConstMem, major,
  minor, textureAlignment, ...
```

Modifying the first CUDA example

- What's needed? The following are necessary elements of a host driver code.
  - Device variable declarations
  - Need to allocate memory on the device
  - Need to transfer host data to the device
  - Need to launch the kernel
  - Need to transfer device data back to the host
  - Need to deallocate space on the device
- Multiple ways of doing this in CUDA Fortran

```
real, device, allocatable, dimension(:) ::
   dev_variable
```

then

```
ierr = cudaMalloc(dev_variable,n)
```

or

```
allocate(dev_variable(n))
```

Modifying the first CUDA example

- What's needed? The following are necessary elements of a host driver code.
  - Device variable declarations
  - Need to allocate memory on the device
  - Need to transfer host data to the device
  - Need to launch the kernel
  - Need to transfer device data back to the host
  - Need to deallocate space on the device
- Multiple ways of doing this in CUDA Fortran

```
real src(n)
real, device, allocatable, dimension(:) ::
  dev_variable
```

then

```
error = cudaMemcpy(dev_variable, src, n)
```

or

```
dev_variable = src
```

Modifying the first CUDA example

- What's needed? The following are necessary elements of a host driver code.
  - Device variable declarations
  - Need to allocate memory on the device
  - Need to transfer host data to the device
  - Need to launch the kernel
  - Need to transfer device data back to the host
  - Need to deallocate space on the device
- Multiple ways of doing this in CUDA Fortran

```
real res(n)
real, device, allocatable, dimension(:) ::
  dev_variable
```

then

```
error = cudaMemcpy(res, dev_variable, n)
```

or

```
res = dev_variable
```

Modifying the first CUDA example

- What's needed? The following are necessary elements of a host driver code.
  - Device variable declarations
  - Need to allocate memory on the device
  - Need to transfer host data to the device
  - Need to launch the kernel
  - Need to transfer device data back to the host
  - Need to deallocate space on the device
- Multiple ways of doing this in CUDA Fortran

```
real, device, allocatable, dimension(:) ::
   dev_variable
```

then

```
error = cudaFree(dev_variable)
```

or

```
deallocate(dev_variable)
```

Modifying the first CUDA example

- **When making direct calls to CUDA API functions, ALWAYS check the error status at the end of a function call.**

```
ierr = cudaFree(dev_variable)
if (ierr.ne.0) then
   print *, “Error in cudaFree for dev_variable
   : “, cudaGetErrorString(ierr)
   stop
else
    print *, “Success in cudaFree!”
endif
```

Modifying the first CUDA example

- What's needed? The following are necessary elements of a host driver code.
  - Device variable declarations
  - Need to allocate memory on the device
  - Need to transfer host data to the device
  - Need to launch the kernel
  - Need to transfer device data back to the host
  - Need to deallocate space on the device

- What's the kernel?
  - Recall that we want to compute the `sqrt()` of a vector element-wise.
  - Where is the parallelism?

- How do I Launch a Kernel?

Your Very First CUDA Kernel

- **Kernels have the following skeleton**

```
attributes(global) subroutine myKernel(a,n)
!
! WHAT NOW??
!
!
end subroutine
```

Your Very First CUDA Kernel

- **Kernels have the following skeleton**

```
attributes(global) subroutine myKernel(src,n)
implicit none
real src(n)
integer, value :: n
integer tx, bx

tx = threadidx%x
bx = blockidx%x

src(tx) = sqrt(src(tx))

end subroutine myKernel
```

Launching Kernels

- **How do I use the Kernel**

```
call myKernel(dev_variable, n)
```

- **What's wrong here?**
  - **Calling the kernel requires a slight modification to the normal Fortran syntax**
  - **Use the "chevron" notation**

```
call myKernel<<<1,n>>>(dev_variable, n)
```

- **Recompile and Run the code**

Launching Kernels

- In the main program in `first.cuf`, set the parameter n to 2000 elements. Recompile and run.

- What happens? Are the results correct? Does anything fail?

- Add the following line before the kernel call

```
print *, maxThreadsDim
```

- Recompile and run.

Launching Kernels

- **How do I fix the code to run for larger vector sizes:**
- **On the host, add**

```
integer numThreads, numBlocks
numThreads = 512
numBlocks = ceiling(n/numThreads)
call myKernel<<<numBlocks,numThreads>>>(dev_variable,n)
```

- **Ok, How Do I fix the Kernel code?**

Launching Kernels

- **Ok, How Do I fix the Kernel code?**

```
attributes(global) subroutine myKernel(src,n)
implicit none
real src(n)
integer, value :: n
integer tx, bx, indx

tx = threadidx%x
bx = blockidx%x
indx = tx + (bx-1)*blockDim%x

src(indx) = sqrt(src(indx))

end subroutine
```

- **Any problems here????**

Launching Kernels

- **Ok, How Do I fix the Kernel code?**

```
attributes(global) subroutine myKernel(src,n)
implicit none
real, dimension(:) :: src
integer, value :: n
integer tx, bx, indx

tx = threadidx%x
bx = blockidx%x
indx = tx + (bx-1)*blockDim%x

If (indx.le.n) then
       src(indx) = sqrt(src(indx))
endif
end subroutine
```

Analyzing Performance

- Recompile and rerun the code. What happens. Do the results look correct?

- Next, let's bump up the size of the vector to something large, say n=1000000. Remove the print statement of the result.

- Let's add some timing code.

```
integer count1, count2, max, rate
real*8 dt

call system_clock ( count1, rate, max)
! Put code in here
call system_clock ( count2, rate, max)
dt = real(count2-count1,kind=8)/real(rate, kind = 8)
print *, "time =", dt, 'second?
```

Analyzing Performance

- **Be sure to write a CPU version of the sqrt() kernel.**

```
do i=1,n
   res(i) = sqrt(src(i))
enddo
```

- **Place the following after the kernel launch**

```
ierr = call cudaThreadSynchronize()
```

- **Place timers around a CPU sqrt computation. Print the time.**
- **Place timers around the GPU computation + the cudaThreadSynchronize() call. Print the time.**
- **Next, extend the timing of the GPU region to include the transfers.**
- **Next, extend the timing of the GPU region to include the allocations.**
- **Next, change the sqrt() kernel to do something like scalar addition.**

3D Wave Equation

- The 3D wave equation with constant wave speed, c, is written

$$\frac{\partial^2 u}{\partial t^2} = c^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}\right)$$

- Finite Difference Time Domain (FDTD) is a typical numerical approach for solving the wave equation.
- Schemes are
  - time-explicit and usually 2nd order in time
  - High-order accuracy in space can easily be achieved:

$$f''(x) = \frac{f(x+\Delta) - 2f(x) + f(x+\Delta)}{\Delta^2} + O(\Delta^2)$$

$$f''(x) = \frac{-f(x-2\Delta) + 16f(x-\Delta) - 30f(x) + 16f(x+\Delta) - f(x+2\Delta)}{12\Delta^2} + O(\Delta^4)$$

3D Wave Equation

- Inserting expansions into wave-equation yields a numerical scheme for computing $u(x,y,z,t) \stackrel{def}{=} u^t_{xyz}$, i.e.

$$u^{t+1}_{x,y,z} = (2 + c_0 c^2 \Delta t^2) u^t_{x,y,z} - u^{t-1}_{x,y,z} + \sum_{i=1}^{k/2} \left[c_{xi}(u^t_{x+i,y,z} + u^t_{x-i,y,z}) + c_{yi}(u^t_{x,y+i,z} + c_{1y} u^t_{x,y-i,z}) + c_{zi}(u^t_{x,y,z+i} + u^t_{x,y,z-i}) \right]$$

- Coefficients are:
  - 2nd Order in space (k=2):

$$c_0 = -2\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right) \qquad c_{x1} = \frac{1}{\Delta x^2}$$

  - 4th Order in space (k=4):

$$c_0 = -\frac{5}{2}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}\right) \qquad c_{xi} = \begin{cases} \frac{4}{3\Delta x^2} & \dots i = 1 \\ -\frac{1}{12\Delta x^2} & \dots i = 2 \end{cases}$$

3D Wave Equation : Stability

- Courant-Friedrich-Lewy Condition (CFL)
  - 2nd Order in space:

$$\Delta t < \frac{c}{\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}}}$$

  - 4th Order in space:

$$\Delta t < \frac{4}{7} \frac{c}{\sqrt{\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2} + \frac{1}{\Delta z^2}}}$$

- If a wave is crossing a discrete grid, then the timestep must be less than the time for the wave to travel adjacent grid points.

- Nasty side effect in 3D: doubling the number of grid cells in the x-direction (cutting the spacing in half), yields a factor of 16 more work (not 8!)

Our Simple Numerical Exercise

- Goal : Develop GPU implementations of the 3D wave equation solver:
- Auxiliary Goal: explore the effect of key device features on the performance of the code.

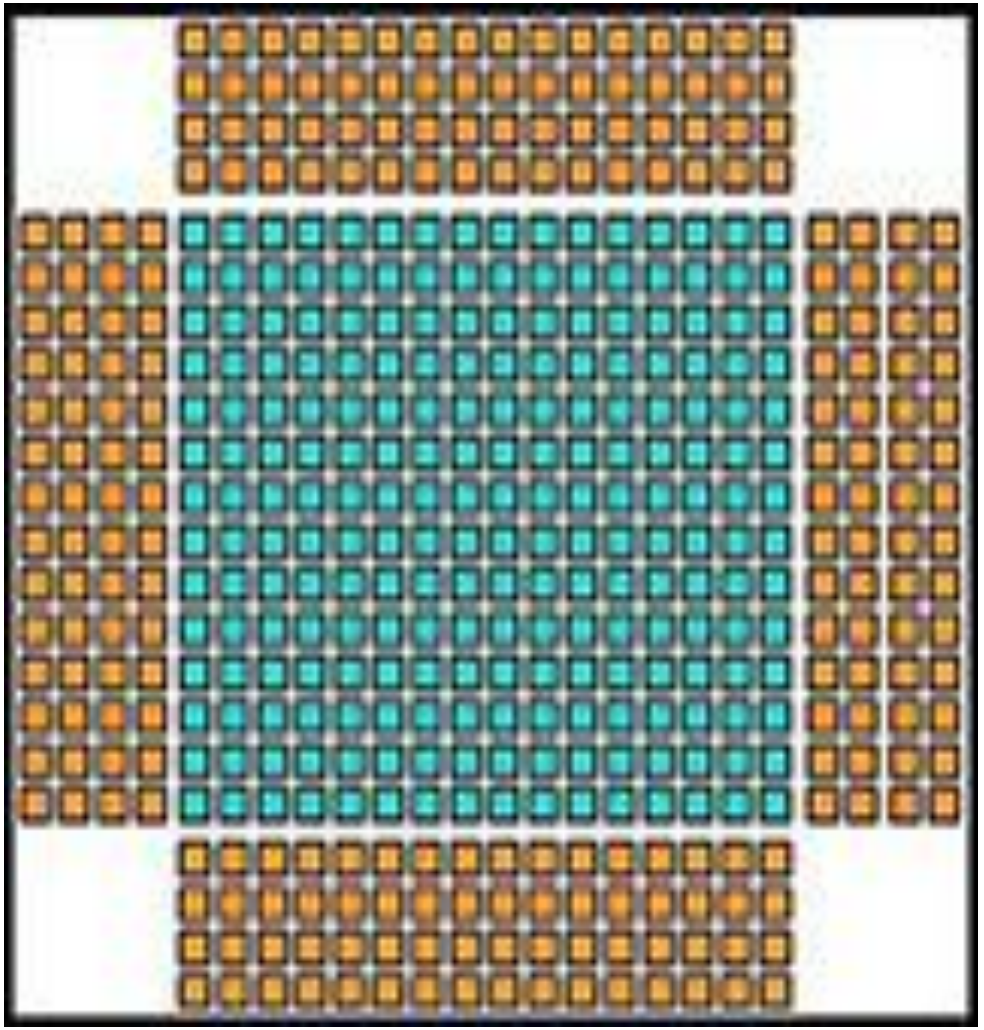
- We will “ignore” certain important aspects of the simulation, including
  - Boundary conditions ... Highly important to deal with these correctly.
  - Correctly, setting the initial conditions ... There are “two” of these here since the time stepping scheme depends on value from the two previous time steps. This is NOT important to deal with from the GPU perspective.
  - No time-dependent sources ... Like boundary conditions, it can be important to deal with these correctly if they are present.

Our Simple Numerical Exercise

- Set the number of Grid cells and time steps at compile time
    - This will likely incur compiler optimizations
- Set the wave speed, c, to 1, without loss of generality
- Initialize the simulation with a delta function –like spike at the center of the domain.
- Since the update requires values from the 2 previous time steps, define 3 buffers: *utp1, ut, utm1* to store the data.

The Physical Domain

- The picture shows an xy-slice of the domain with 4 ghost cells in each direction. 8th order in space.
- In our code, we will consider only 2nd and 4th order in space.
- Thus, our domains will have 1 or 2 ghost (guard) cells respectively.
- In our code, x is the fastest varying dimension followed by y and z. This has important considerations for the device kernels.



Code Structure

driver.CUF

- Initializes simulation parameters and storage buffers
- Calls fdtdCPU/GPUDriver subroutines
- Calls timing functions for assessing speedup
- Checks results … always do this rigorously!!!!

fdtdCPU.CUF

- triple buffered time-stepping
- OpenMP directives included around the core loop

fdtdGPU.CUF

- currently has 3 waveEqUpdateDriver subroutines (host code)
- each subroutine uses distinct CUDA Fortran features.
- triple buffered time-stepping in the waveEqUpdateDriver subroutines.

CPU Code Structure (with OpenMP directives)

```
!$OMP   PARALLEL DO &
!$OMP   DEFAULT(SHARED) &
!$OMP   PRIVATE(i, j, k, p)
      do k=1+Nghost,Nz+Nghost
        do j=1+Nghost,Ny+Nghost
          do i=1+Nghost,Nx+Nghost
            unew(i,j,k) = -uold(i,j,k) + (2+dt2*c0)*ucurrent(i,j,k)
            do p=1,Nghost
              unew(i,j,k)=unew(i,j,k)       &
                        + dt2*c1x(p)*(ucurrent(i+p,j,k)+ucurrent(i-p,j,k)) &
                        + dt2*c1y(p)*(ucurrent(i,j+p,k)+ucurrent(i,j-p,k)) &
                        + dt2*c1z(p)*(ucurrent(i,j,k+p)+ucurrent(i,j,k-p))
            enddo
          enddo
        enddo
      enddo
!$OMP   END PARALLEL DO
```

GPU Implementation 1

- Driver code : waveEqUpdateGPUDriver1
  - Data allocation and freeing
  - Data transfer from CPU-GPU and vice versa
  - Kernel setup parameters
  - Time stepping and kernel invocation

- Kernel : waveEqUpdateGPUKernel1
  - Vectorized kernel code
  - Use of constant device memory for coefficients
  - Array indexing via thread and block values

waveEqUpdateGPUDriver1

```
subroutine waveEqUpdateGPUDriver1(Nx,Ny,Nz,Nghost,Nt,u0,u1,u2,dt)
implicit none
!-----input
integer Nx,Ny,Nz,Nghost,Nt
real dt
real u0(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost)
real u1(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost)
real u2(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost)
!-----temporary device arrays and data
real, device, allocatable, dimension(:,:,:) :: dev_u0, dev_u1, dev_u2
type(dim3) :: dimGrid, dimBlock
integer it
!-----allocations
allocate(dev_u0(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost))
allocate(dev_u1(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost))
allocate(dev_u2(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost))
!-----transfer ... using overloaded = operator
dev_u0 = u0(1:Nx+2*Nghost,1:Ny+2*Nghost,1:Nz+2*Nghost)
dev_u1 = u1(1:Nx+2*Nghost,1:Ny+2*Nghost,1:Nz+2*Nghost)
dev_u2 = u2(1:Nx+2*Nghost,1:Ny+2*Nghost,1:Nz+2*Nghost)
```

waveEqUpdateGPUDriver1

```
subroutine waveEqUpdateGPUDriver1(Nx,Ny,Nz,Nghost,Nt,u0,u1,u2,dt)
implicit none
…
!-----transfer ... using overloaded = operator
dev_u0 = u0(1:Nx+2*Nghost,1:Ny+2*Nghost,1:Nz+2*Nghost)
dev_u1 = u1(1:Nx+2*Nghost,1:Ny+2*Nghost,1:Nz+2*Nghost)
dev_u2 = u2(1:Nx+2*Nghost,1:Ny+2*Nghost,1:Nz+2*Nghost)
!-----thread, block configuration
dimGrid = dim3(Ny,Nz,1)
dimBlock = dim3(Nx,1,1)
! Time-stepping with kernel calls.
do it=1,Nt
   if (mod(it,3).eq.1) then
     call waveEqUpdateGPUKernel1<<<dimGrid,dimBlock>>>(Nx,Ny,Nz,Nghost,dev_u0,dev_u1,dev_u2,dt)
   elseif (mod(it,3).eq.2) then
     call waveEqUpdateGPUKernel1<<<dimGrid,dimBlock>>>(Nx,Ny,Nz,Nghost,dev_u1,dev_u2,dev_u0,dt)
   else
      call waveEqUpdateGPUKernel1<<<dimGrid,dimBlock>>>(Nx,Ny,Nz,Nghost,dev_u2,dev_u0,dev_u1,dt)
   endif
enddo
```

waveEqUpdateGPUDriver1

```
subroutine waveEqUpdateGPUDriver1(Nx,Ny,Nz,Nghost,Nt,u0,u1,u2,dt)
implicit none
…
! Time-stepping with kernel calls.
do it=1,Nt
   if (mod(it,3).eq.1) then
     call waveEqUpdateGPUKernel1<<<dimGrid,dimBlock>>>(Nx,Ny,Nz,Nghost,dev_u0,dev_u1,dev_u2,dt)
   elseif (mod(it,3).eq.2) then
     call waveEqUpdateGPUKernel1<<<dimGrid,dimBlock>>>(Nx,Ny,Nz,Nghost,dev_u1,dev_u2,dev_u0,dt)
   else
      call waveEqUpdateGPUKernel1<<<dimGrid,dimBlock>>>(Nx,Ny,Nz,Nghost,dev_u2,dev_u0,dev_u1,dt)
   endif
enddo
!-----transfer back... using overloaded = operator
u0(1:Nx+2*Nghost,1:Ny+2*Nghost,1:Nz+2*Nghost) = dev_u0
u1(1:Nx+2*Nghost,1:Ny+2*Nghost,1:Nz+2*Nghost) = dev_u1
u2(1:Nx+2*Nghost,1:Ny+2*Nghost,1:Nz+2*Nghost) = dev_u2
!-----deallocations
deallocate(dev_u0, dev_u1, dev_u2)
end subroutine waveEqUpdateGPUDriver1
```

waveEqUpdateGPUKernel1 : Device Code

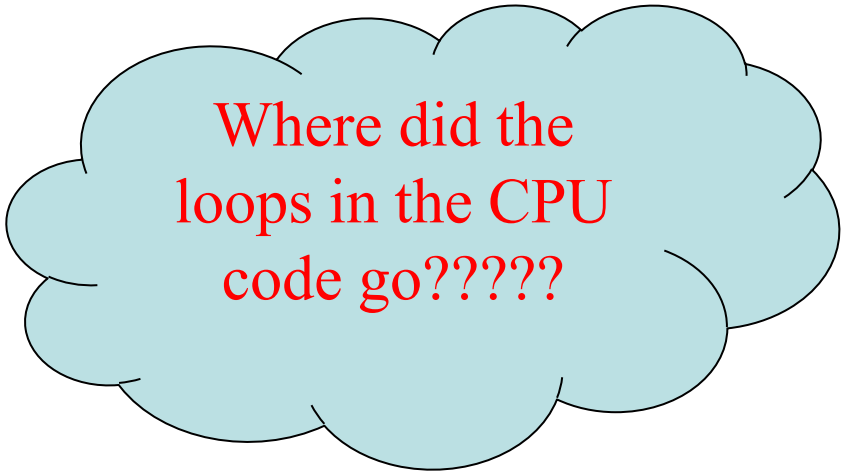
```
attributes(global) subroutine
waveEqUpdateGPUKernel1(Nx,Ny,Nz,Nghost,u0,u1,u2,dt)

implicit none

integer, value :: Nx, Ny, Nz, Nghost
real :: u0(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost)
real :: u1(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost)
real :: u2(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost)

real, value :: dt
integer p, by, bz, tx, indx, indy, indz
real dt2, temp_u2
tx = threadidx%x
by = blockidx%x
bz = blockidx%y
```

**** Recall : the driver (host code) function had variables as ***

```
real, device, allocatable, dimension(:,:,:) :: dev_u0, dev_u1, dev_u2
```

waveEqUpdateGPUKernel1 : Device Code

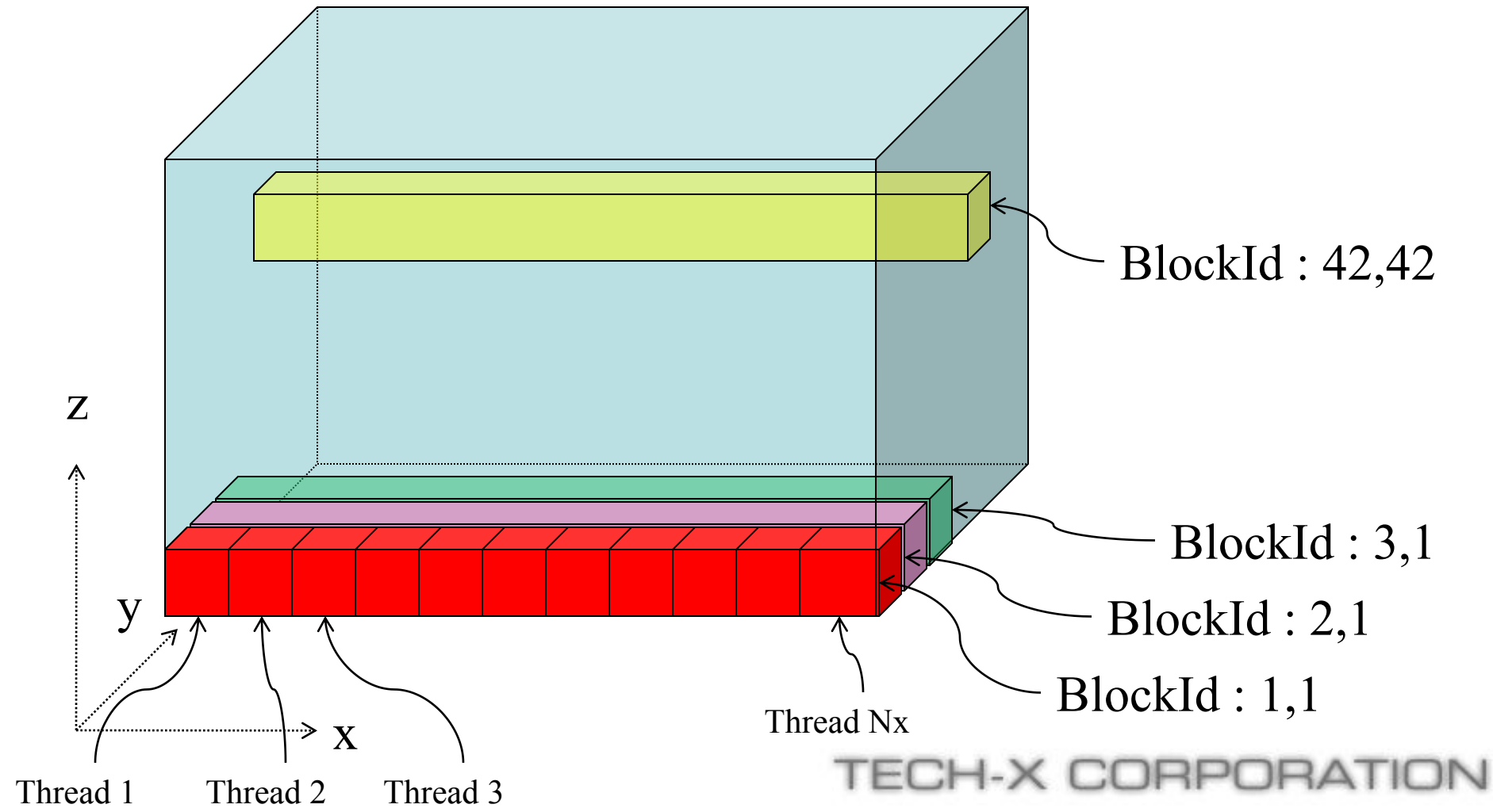
```
attributes(global) subroutine waveEqUpdateGPUKernel1(Nx,Ny,Nz,Nghost,u0,u1,u2,dt)
implicit none
…
tx = threadidx%x
by = blockidx%x
bz = blockidx%y
indx = tx + Nghost
indy = by + Nghost
indz = bz + Nghost
! Only compute in the physical region
if (tx.gt.Nghost .and. tx.le.Nx+Nghost) then
  dt2 = dt*dt
  temp_u2 = -u0(indx,indy,indz) + (2.+dt2*dev_c0)*u1(indx,indy,indz)
  do p=1,Nghost
    temp_u2 = temp_u2 &
      + dt2*dev_c1x(p)*(u1(indx+p,indy,indz)+u1(indx-p,indy,indz)) &
      + dt2*dev_c1y(p)*(u1(indx,indy+p,indz)+u1(indx,indy-p,indz)) &
      + dt2*dev_c1z(p)*(u1(indx,indy,indz+p)+u1(indx,indy,indz-p))
  enddo
  u2(indx,indy,indz) = temp_u2
endif
end subroutine waveEqUpdateGPUKernel1
```



Parallelization Choices in the Grid and Block Configuration.

```
!----thread, block configuration
dimGrid = dim3(Ny,Nz,1)
dimBlock = dim3(Nx,1,1)
```

The parallelization is contained in the grid and block configuration!!!!



Where did the Loops Go?

```
!----thread, block configuration
dimGrid = dim3(Ny,Nz,1)
dimBlock = dim3(Nx,1,1)
```

The parallelization is contained in the grid and block configuration!!!!

Y and Z loops are contained in the dimGrid configuration. The Loop in x is contained in the dimBlock configuration.

```
tx = threadidx%x
by = blockidx%x
bz = blockidx%y
indx = tx + Nghost
indy = by + Nghost
indz = bz + Nghost
```

Thread and block indices play the analogous of loop indices in CPU code

```
! Only compute in the physical region
temp_u2 = -u0(indx,indy,indz) + (2.+dt2*dev_c0)*u1(indx,indy,indz)
  do p=1,Nghost
    temp_u2 = temp_u2 &
      + dt2*dev_c1x(p)*(u1(indx+p,indy,indz)+u1(indx-p,indy,indz)) &
      + dt2*dev_c1y(p)*(u1(indx,indy+p,indz)+u1(indx,indy-p,indz)) &
      + dt2*dev_c1z(p)*(u1(indx,indy,indz+p)+u1(indx,indy,indz-p))
  enddo
```

waveEqUpdateGPUKernel1 : Device Code Good Practices

- If a variable is repeatedly updated by a single thread, then
  - USE register variables
  - Write to Global Memory Once!!!!!
- Highly important on TESLA 1060 series (and earlier) GPUs

Possibly not as important FERMI

```
! Only compute in the physical region
if (tx.gt.Nghost .and. tx.le.Nx+Nghost) then
   dt2 = dt*dt
   temp_u2 = -u0(indx,indy,indz) + (2.+dt2*dev_c0)*u1(indx,indy,indz)
   do p=1,Nghost
      temp_u2 = temp_u2 &
         + dt2*dev_c1x(p)*(u1(indx+p,indy,indz)+u1(indx-p,indy,indz)) &
         + dt2*dev_c1y(p)*(u1(indx,indy+p,indz)+u1(indx,indy-p,indz)) &
         + dt2*dev_c1z(p)*(u1(indx,indy,indz+p)+u1(indx,indy,indz-p))
   enddo
   u2(indx,indy,indz) = temp_u2
endif
```

waveEqUpdateGPUKernel1 : Device Code Good Practices

What's this???

dev_c0, dev_c1x, dev_c1y, dev_c1z were NOT declared in the function or in the signature.

```
! Only compute in the physical region
if (tx.gt.Nghost .and. tx.le.Nx+Nghost) then
   dt2 = dt*dt
   temp_u2 = -u0(indx,indy,indz) + (2.+dt2*dev_c0)*u1(indx,indy,indz)
   do p=1,Nghost
      temp_u2 = temp_u2 &
         + dt2*dev_c1x(p)*(u1(indx+p,indy,indz)+u1(indx-p,indy,indz)) &
         + dt2*dev_c1y(p)*(u1(indx,indy+p,indz)+u1(indx,indy-p,indz)) &
         + dt2*dev_c1z(p)*(u1(indx,indy,indz+p)+u1(indx,indy,indz-p))
   enddo
   u2(indx,indy,indz) = temp_u2
endif
```

waveEqUpdateGPUKernel1 : Device Code Using Constant data

- In the module fdtdGPU.CUF, we do

```
module fdtdGPU

use cudafor

!-----coefficients are stored in constant data
real, constant :: dev_c0
real, constant :: dev_c1x(2), dev_c1y(2), dev_c1z(2)

contains
…
```

- In the fdtdGPUDriver subroutine, we do

```
!-----coefficients are copied to constant data
dev_c0  = c0
dev_c1x = c1x
dev_c1y = c1y
dev_c1z = c1z
```

waveEqUpdateGPUKernel1 : Device Code Best Practices Using Constant data

- Use constant data for small sets of data reusable across all thread blocks
- Constant data can only be altered by a **Host** subroutine
- Constant data can be used for computation in host subroutines
  - **DO NOT DO THIS!**
  - Why? The data lives on the device. If you attempt to use it on the host, each call will issue a memcpy operation from device to host. **BAD!!!!**

BEWARE Module Variables!!!!

- In the module fdtdGPU.CUF, we do

```
module fdtdGPU

use cudafor

!-----coefficients are stored in constant data
real, constant :: dev_c0
real, constant :: dev_c1x(2), dev_c1y(2), dev_c1z(2)
```

- If you want to do parallel-GPU computation with a single CPU core controlling a single GPU, then one must set the device for each rank via the following (in driver.CUF)

```
cudaError = cudaSetDevice(myCpuRank)
```

** **Problem** ** The call to cudaSetDevice **MUST** be the first cuda API call that executes code on the device.

BEWARE Module Variables!!!!

- In the module fdtdGPU.CUF, we do

```
module fdtdGPU

use cudafor

!-----coefficients are stored in constant data
real, constant :: dev_c0
real, constant :: dev_c1x(2), dev_c1y(2), dev_c1z(2)
```

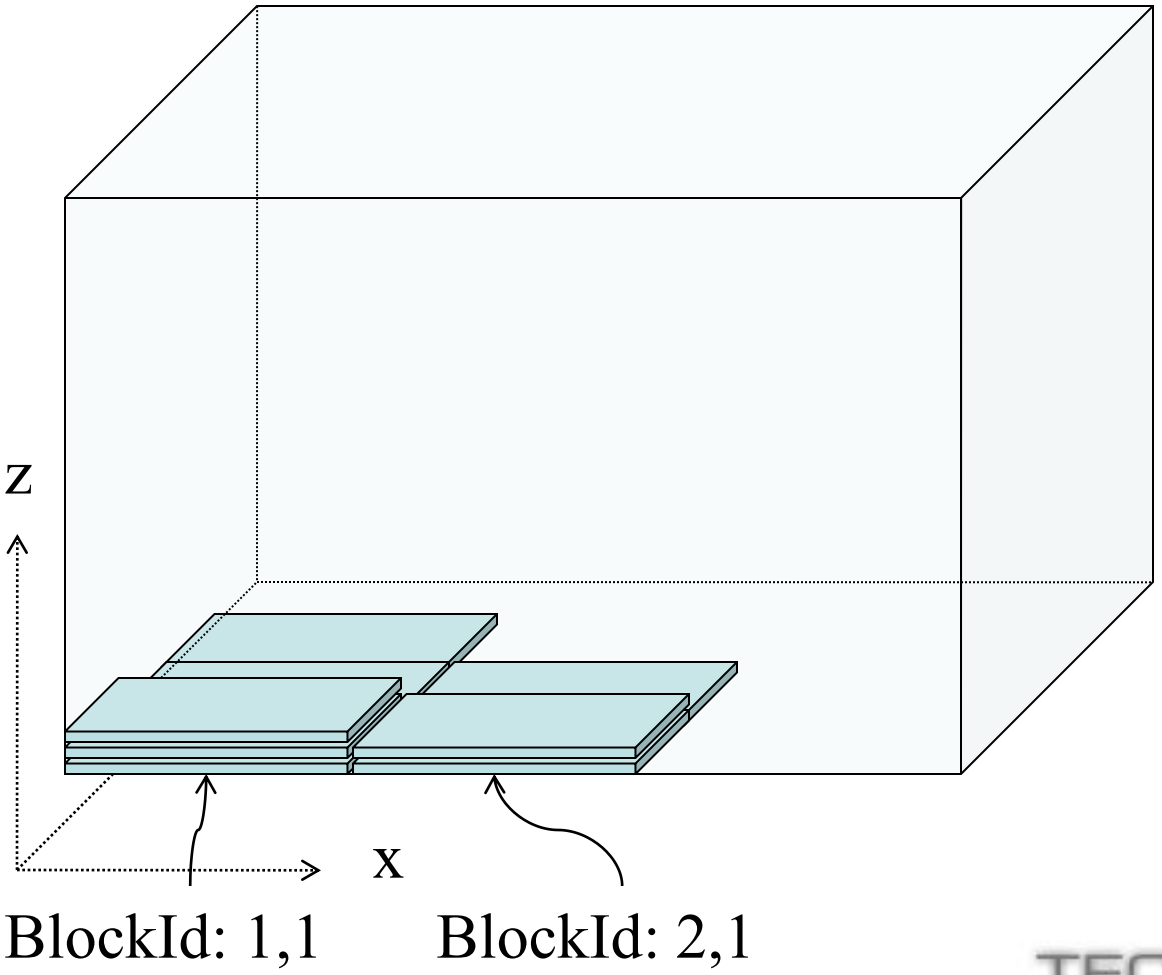
- In the FDTD Program, we do

```
Program FDTD_3D

  use cudafor
  use fdtdGPU
  use fdtdCPU
  implicit none
  …
  cudaError = cudaSetDevice(myCpuRank)
```

•This code performs allocations on the device!!
• cudaSetDevice does not happen first!!!

Other Parallelization Strategies

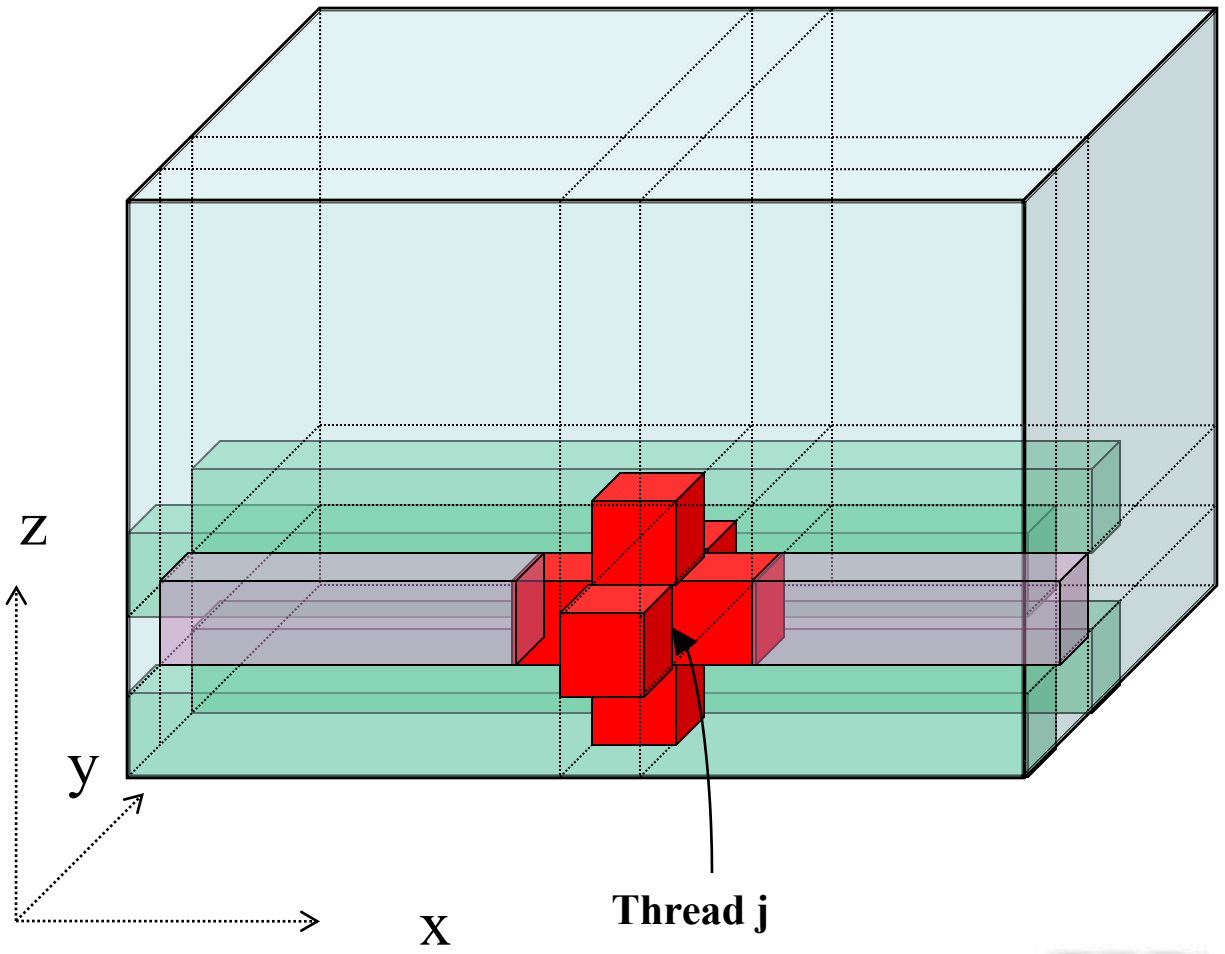
- P. Micikevicius. “3D Finite Difference Computation on GPUs using CUDA,” Proceedings of Supercomputing 2009.



- Each panel is a 16x16 (or 32x32) grid of cells in a plane parallel to the xy plane and a single cell in z
- A single warp (thread block) does the computation for one of these panels.
- Strategy “could” benefit from advanced device features

GPU Implementation 2

- Key to good performance : Memory coalescence and Minimize reads/write to global memory!



- For coalescence, consecutive threads must read adjacent slots memory
- Ideally, each thread block will read a multiple of 16 or 32 memory slots in a single instruction.
- For arbitrary size domain in x, this does not happen!
- Solution, pad the memory in the fast varying dimension.

GPU Implementation 2

- Not all memory accesses can benefit from coalescence

$$u_{x,y,z}^{t+1} = (2 + c_0 c^2 \Delta t^2) u_{x,y,z}^{t} - u_{x,y,z}^{t-1} + \sum_{i=1}^{k/2} \left[c_{xi} (u_{x+i,y,z}^{t} + u_{x-i,y,z}^{t}) + c_{yi} (u_{x,y+i,z}^{t} + c_{1y} u_{x,y-i,z}^{t}) + c_{zi} (u_{x,y,z+i}^{t} + u_{x,y,z-i}^{t}) \right]$$

- Typically, one attempts to get coalescence for memory reads/writes parallel to the fastest varying dimension (x)
- For this computation, all reads should be coalesced provided we pad/pitch the memory in the x-direction
- waveEqUpdateGPUDriver2/Kernel2

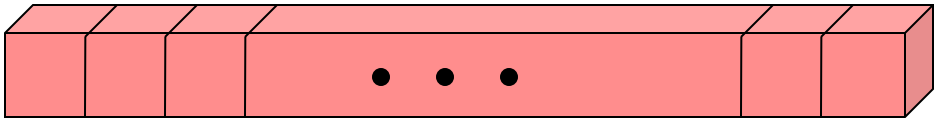
waveEqUpdateGPUDriver2 : cudaMallocPitch

```
subroutine waveEqUpdateGPUDriver2(Nx,Ny,Nz,Nghost,Nt,u0,u1,u2,dt)
implicit none
!-----input
integer Nx,Ny,Nz,Nghost,Nt
real dt
real u0(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost)
real u1(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost)
real u2(Nx+2*Nghost,Ny+2*Nghost,Nz+2*Nghost)
!-----temporary device arrays and data
real, device, allocatable, dimension(:,:) :: dev_u0, dev_u1, dev_u2
type(dim3) :: dimGrid, dimBlock
integer it
integer pitch, width, height, error0, error1, error2
!-----allocations via cudaMallocPitch
error0 = cudaMallocPitch(dev_u0, pitch, Nx+2*Nghost, (Ny+2*Nghost)*(Nz+2*Nghost))
error1 = cudaMallocPitch(dev_u1, pitch, Nx+2*Nghost, (Ny+2*Nghost)*(Nz+2*Nghost))
error2 = cudaMallocPitch(dev_u2, pitch, Nx+2*Nghost, (Ny+2*Nghost)*(Nz+2*Nghost))
!-----error checking
if (error0.ne.0.or. error1.ne.0.or error2.ne.0) then
   print *, “Something BAD Happened!!!!“
   stop
endif
```

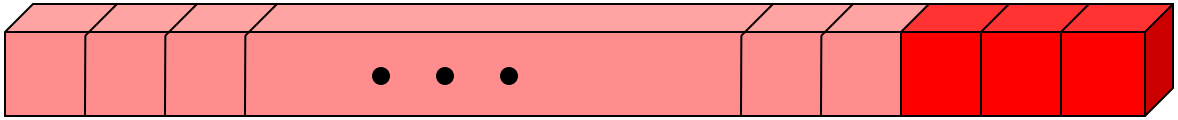
waveEqUpdateGPUDriver2 : cudaMallocPitch

```
!-----allocations via cudaMallocPitch
error0 = cudaMallocPitch(dev_u0, pitch, Nx+2*Nghost, (Ny+2*Nghost)*(Nz+2*Nghost))
```

- What is going on in this code?
- In the first driver code, we used the **allocate** statement. Using **allocate**, a contiguous chunk of memory is a line in x-direction for fixed y and fixed z.



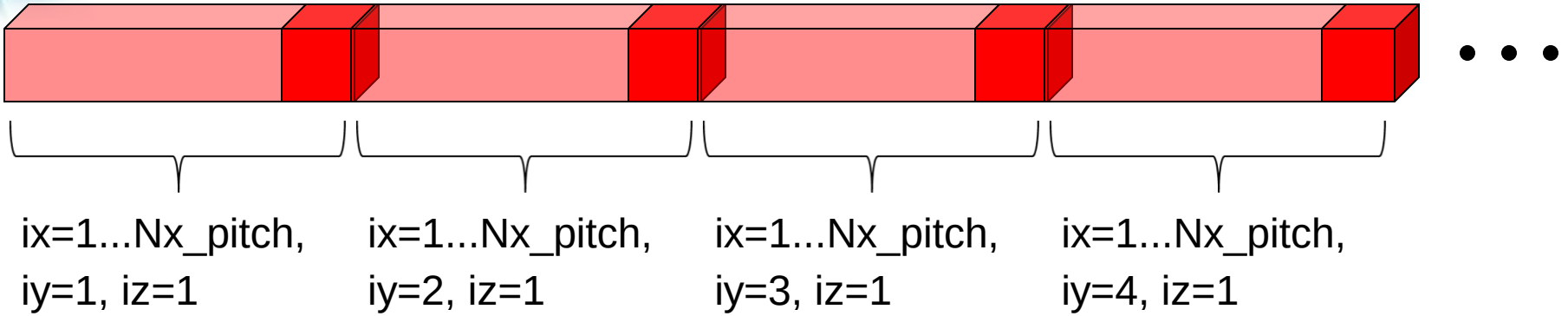
- Using **cudaMallocPitch**, a contiguous chunk of memory is a padded-line in the x-direction for fixed x and y.



- Pitch depends on the model of the GPU. On FERMI, arrays are padded/pitched to multiples of 128.

waveEqUpdateGPUDriver2 : cudaMallocPitch

- Linear memory will have the following structure:



- Red cells are unphysical ... Padded/Pitched region.
- cudaMallocPitch produces a 2D data structure.

  *** Slower varying dimensions are stacked to fake 3D ***
- Ideally, use cudaMalloc3D to get a 3D a pitched 3D data structure. Was not working correctly in PGI 10.4. Should be fixed in 10.

  *** Challenging to use cudaMalloc3D, cudaMemcpy3D ***

waveEqUpdateGPUDriver2 : cudaMemcpy2D

3D Host memory needs to be mapped into 2D pitched device memory

```
!-----transfer from 3D Host to the 2D pitched memory ... requires care
call cudaMemcpy3DHostTo2DDevice(dev_u0, pitch, u0, Nx+2*Nghost, &
     Nx+2*Nghost, Ny+2*Nghost, Nz+2*Nghost)


!----a subroutine for doing 3D → 2D mapping (and the reverse)
subroutine cudaMemcpy3DHostTo2DDevice(dstPtr, dstPitch, &
                srcPtr, srcPitch, width, height, depth)
implicit none
!     input parameters
real, device, allocatable, dimension(:,:) :: dstPtr
real, dimension(:,:,:) :: srcPtr
integer :: dstPitch, srcPitch, width, height, depth
!     cudaMemcpy2D
error = cudaMemcpy2D(dstPtr, dstPitch, srcPtr, srcPitch, &
     width, height*depth, cudaMemcpyHostToDevice)
```

waveEqUpdateGPUDriver2

```
subroutine waveEqUpdateGPUDriver2(Nx,Ny,Nz,Nghost,Nt,u0,u1,u2,dt)
implicit none
…
!-----thread, block configuration
dimGrid = dim3(Ny,Nz,1)
dimBlock = dim3(pitch,1,1)
! Time stepping loop
do it=1,Nt
   if (mod(it,3).eq.1) then
      call
waveEqUpdateGPUKernel2<<<dimGrid,dimBlock>>>(Nx,Ny,Nz,Nghost,pitch,dev_u0,dev_u1,dev_u2,dt)
   elseif (mod(it,3).eq.2) then
      call
waveEqUpdateGPUKernel2<<<dimGrid,dimBlock>>>(Nx,Ny,Nz,Nghost,pitch,dev_u1,dev_u2,dev_u0,dt)
   else
      call
waveEqUpdateGPUKernel2<<<dimGrid,dimBlock>>>(Nx,Ny,Nz,Nghost,pitch,dev_u2,dev_u0,dev_u1,dt)
   endif
enddo
!-----transfer back from 2D pitched memory to 3D Host ... requires care
call cudaMemcpy2DDeviceTo3DHost(u2, Nx+2*Nghost, dev_u2, pitch, &
      Nx+2*Nghost, Ny+2*Nghost, Nz+2*Nghost)
!-----deallocations via cudaFree … same for dev_u0, dev_u1
error2 = cudaFree(dev_u2)
```

Each thread block has pitch threads!

waveEqUpdateGPUKernel2

```
attributes(global) subroutine waveEqUpdateGPUKernel2(Nx,Ny,Nz,Nghost,Nt,u0,u1,u2,dt)
implicit none
integer, value :: Nx, Ny, Nz, Nghost, pitch
real :: u0(pitch, (Ny+2*Nghost)*(Nz+2*Nghost))
real :: u1(pitch, (Ny+2*Nghost)*(Nz+2*Nghost))
real :: u2(pitch, (Ny+2*Nghost)*(Nz+2*Nghost))
real, value :: dt
integer p, by, bz, tx, ind_y_z, ind_ymp_z, ind_ypp_z , ind_y_zmp, ind_y_zpp
integer NyExt
real dt2, temp_u2

tx = threadidx%x
by = blockidx%x
bz = blockidx%y
NyExt = Ny+2*Nghost
ind_y_z = by + Nghost + (bz + Nghost - 1)*NyEx
```

Arrays are now 2D in the kernel!

waveEqUpdateGPUKernel2

```
attributes(global) subroutine waveEqUpdateGPUKernel2(Nx,Ny,Nz,Nghost,Nt,u0,u1,u2,dt)
Implicit none
…
tx = threadidx%x
by = blockidx%x
bz = blockidx%y
NyExt = Ny+2*Nghost
ind_y_z = by + Nghost + (bz + Nghost - 1)*NyExt
! Do not compute in the ghost cells or the pitched region
if (tx.gt.Nghost .and. tx.le.(Nx+Nghost)) then
   dt2 = dt*dt
   temp_u2 = -u0(tx, ind_y_z)+ (2.+dt2*dev_c0)*u1(tx, ind_y_z)
   do p=1,Nghost
      ind_ymp_z = ind_y_z-p
      ind_ypp_z = ind_y_z+p
      ind_y_zmp = ind_y_z-p*NyExt
      ind_y_zpp = ind_y_z+p*NyExt
      temp_u2 = temp_u2 &
        + dt2*dev_c1x(p)*(u1(tx+p, ind_y_z)+u1(tx-p, ind_y_z)) &
              + dt2*dev_c1y(p)*(u1(tx, ind_ypp_z)+u1(tx, ind_ymp_z)) &
              + dt2*dev_c1z(p)*(u1(tx, ind_y_zpp)+u1(tx, ind_y_zmp))
   enddo
   u2(tx, ind_y_z) = temp_u2
endif
```

Indexing variables into 2D arrays!

Kernel Comparisons

- On Tesla C1060, Kernel2 gives 75% improvement over Kernel1. On FERMI, GTX 480, Kernel2 gives 30-40% improvement over Kernel1
- For large problems, Kernel2 is roughly 60X faster than the CPU version. Kernel1 is roughly 40X faster. CPU on the test machine is NOT state of the art. Nor is the code particulalry good.
- Keep in mind, speedup is a loaded and hence "volatile" number.
- A better assessment of performance would be to measure how close am I to peak theoretical performance in terms of
    - Memory bandwidth
    - Block occupancy

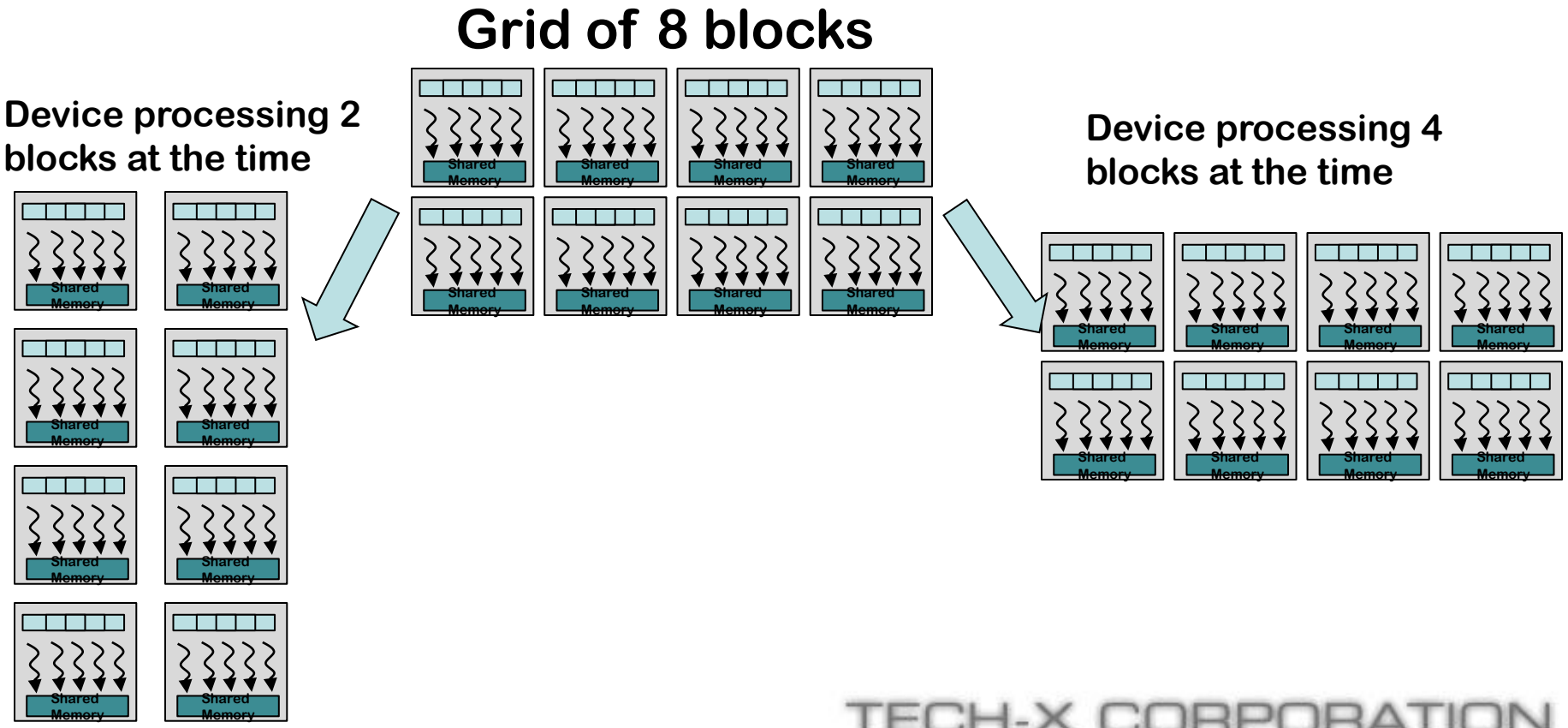
Optimization 3

- **Recall the keys to good performance : Memory coalescence and minimize reads/write to global memory!**
- **Addressed the first key via cudaMallocPitch. In 3D, cudaMalloc3D will achieve the same end.**
- **What else can be done here?**

```
if (tx.gt.Nghost .and. tx.le.(Nx+Nghost)) then
  dt2 = dt*dt
  temp_u2 = -u0(tx, ind_y_z)+ (2.+dt2*dev_c0)*u1(tx, ind_y_z)
  do p=1,Nghost
    ind_ymp_z = ind_y_z-p
    ind_ypp_z = ind_y_z+p
    ind_y_zmp = ind_y_z-p*NyExt
    ind_y_zpp = ind_y_z+p*NyExt
    temp_u2 = temp_u2 &
      + dt2*dev_c1x(p)*(u1(tx+p, ind_y_z)+u1(tx-p, ind_y_z)) &
      + dt2*dev_c1y(p)*(u1(tx, ind_ypp_z)+u1(tx, ind_ymp_z)) &
      + dt2*dev_c1z(p)*(u1(tx, ind_y_zpp)+u1(tx, ind_y_zmp))
  enddo
  u2(tx, ind_y_z) = temp_u2
endif
```

Parallel Scalability via Grid of Thread Blocks

- Kernels are launched as **Grid** of thread blocks
- GPU executes multiple blocks concurrently
- No execution order defined for individual blocks
- Enables scalability without code rewrite

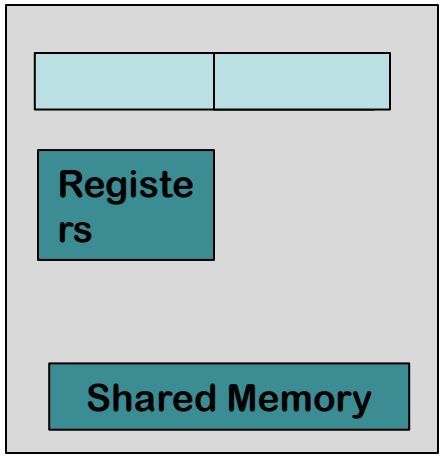


CUDA “Device Architecture”

- GPU : collection of Streaming Multiprocessors (SM)
  - Basically vector processors, 1 - 15 SM per GPU
- A block is executed on a single SM
  - Multiple blocks execute on same SM concurrently, share resources
  - No block migration

- Registers
- Shared Memory
- Global Memory

- Host access only to global memory



CUDA Programming

- Basics for getting code to run on GPU:
  - GPU Memory management
  - GPU Kernel Launches
  - Kernel design
- Special considerations
  - Synchronization
  - Error checking
- Debugging
- Optimization
  - Will be covered in next session

Programming Languages for CUDA

- Native CUDA interface is C++ (used to be C)
- Fortran accessible in two ways:
  - Mixed-language programming
  - PGI compiler (more vendors are following)
- Fortran via Mixed Language
  - Calling C functions from Fortran
  - Watch out for calling conventions/name mangling
  - No (additional) vendor lock-in
- Portland Group (PGI) compiler
  - CUDA Fortran interface
  - Directive based GPU programming (high-level)
  - Proprietary

Memory Management

- CPU and GPU have separate memory spaces
- CPU is in charge of managing GPU memory
  - allocate/free
  - data transfer to/from GPU
- Physical addresses on GPU
- Memory allocation

```
float *devPtr;
cudaMalloc(&devPtr, 100);
cudaFree(devPtr);

real, allocatable, device :: v(:)
istat = cudaMalloc(v, 100)
istat = cudaFree(v)
```

- PGI declares memory location

Data Transfer between CPU and GPU

```
cudaMemcpy(void *dst, void *src, size_t nbytes,
           enum cudaMemcpyKind direction);
```

- **direction specifies location of src and dst**
  cudaMemcpyHostToDevice, cudaMemcpyDeviceToHost, cudaMemcpyHostToHost, cudaMemcpyDeviceToDevice

- **Blocks CPU thread**
- **Does not start until kernel completed**

- **PGI Fortran does not require 'direction'**

```
real, allocatable, device :: dst(1024)
real src(512)
istat = cudaMemcpy(dst, src, 512)
```

Kernels

- **Kernel: Subroutine executed on the device**

- **Some restrictions**
  - **Can only access GPU memory**
  - **Non recursive, no optional arguments, no static data members**

- **Kernel defined via** `attribute (global)`

```
attribute(global) subroutine sqrtKernel(n, a)
  real, dimension(*) :: a
  integer, value :: n
  ...
end subroutine sqrtKernel
```

Determining unique thread identifier

- **Grid configuration can be 1D, 2D or 3D**
  - **Both for blocks and threads**
  - **Simplifies operations on higher dimensional arrays**

- **Block and thread index structs**

```
blockidx % x, threadidx % y, ..
```

- **Threads can compute their unique index**

```
tid = (blockidx % x – 1) * blockdim % x +  threadidx % x
```

Launching Kernels

- **Called from CPU with thread grid configuration**

```
call sqrtKernel<<<blockPerGrid, threadsPerBlock>>>(n, a)
```

- **Grid dimensions**
  - **Block per grid, Thread per blocks**
  - **Can be scalar or three element**

```
type(dim3) :: dimGrid, dimBlock
dimGrid = dim3(N/16, M/16, 1)
dimBlock = dim3(16, 16, 1)

call complexKernel<<< dimGrid, dimBlock >>> (n, u, v)
```

Example 1: Skeleton and Data Transfer

```
program main
   use cudafor
   integer ierr, i

   real, device:: dst(20)
   real src(10), res(10)

   do i=1,10 src(i) = i

   ierr = cudaMemcpy(dst, src, 10)


   ierr = cudaMemcpy(res, dst, 10)

   print *, res
 end
```

PGI CUDA module

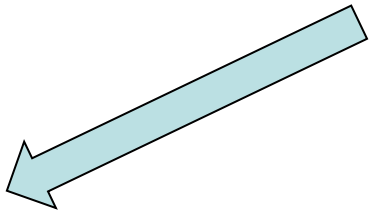
Allocation on GPU

Transfer to/from GPU

Example 1 (cont): Kernel Launch

```
program main
   use cudafor
   integer ierr, i

   real, device:: dst(20)
   real src(10), res(10)

   do i=1,10 src(i) = i

   ierr = cudaMemcpy(dst, src, 10)

   call sqrtKernel<<<1, 10>>>(10, dst)

   ierr = cudaMemcpy(res, dst, 10)

   print *, res
 end
```

Launching the kernel, 1 block, 10 threads per block



Example 1 (cont): Kernel

```
attribute(global) subroutine sqrtKernel(n, a)
      real, dimension(*) :: a
      integer, value :: n
      integer, value :: i

      i = (blockidx%x - 1) * blockdim%x + threadidx%x
      if( i <= n )  a(i) = sqrt( a(i) )

end subroutine sqrtKernel

 program main
    ...

    call sqrtKernel<<<1, 10>>>(10, dst)
    ....
    print *, res
  end
```

Compiling CUDA Programs

- **PGI / Fortran**
  - .cuf/.CUF extension to enable CUDA

```
pgCUF test.cuf
```

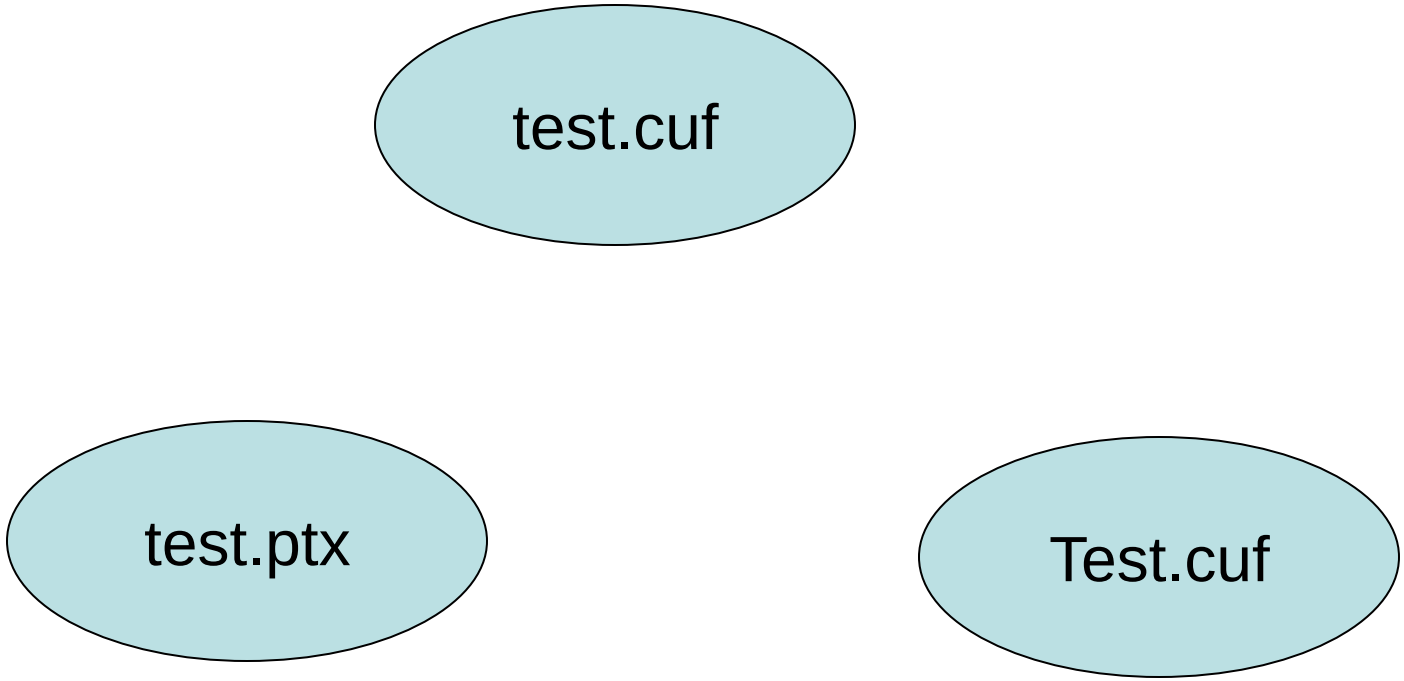
  - Selecting a specific CUDA version

```
pgCUF -ta=cuda2.3  test.cuf
pgCUF -ta=cc20     test.cuf
pgCUF -Mcuda=emu    test.CUF
```

- **C/C++**
  - nvcc compiler wrapper, uses Visual C++/gcc under the hood

```
nvcc -arch=sm_20 test.c
```

Where is the GPU binary?



Box-Car Filter, Part 1

```
attribute(global) subroutine boxCar(n, in, out)
     real, dimension(*) :: in
     real, dimension(*) :: out
     integer, value :: n, i

     i = (blockidx%x - 1) * blockdim%x + threadidx%x

     if( i > 1 and i < n ) then
        out(i) =  (in(i-1) + 2 * in(i) + in(i+1)) / 4.
      end if

end subroutine boxCar
```

Each thread 4 global memory accesses!

Shared Memory

- **Limited amount of memory shared among all threads in a block**
  - **Some 16kB – 48 kB per multi-processor**
- **Very fast access by threads**
  - **Comparable to registers**
- **Explicitly sized**

```
real, shared, dimension(16) :: sharedMem
```

- **Implicitly sized**

```
real, shared, dimension(*) :: sharedMem

call sqrtKernel<<<1, 10, 16>>>(10, dst)
```

Box-Car Filter, Part 2

```
attribute(global) subroutine boxCar(n, in, out)
     real, dimension(*) :: in
     real, dimension(*) :: out
     real, shared, dimension(*) :: tmp
     integer, value :: n, i

     i = (blockidx%x - 1) * blockdim%x + threadidx%x

    tmp[i] = in[i]

     if( i > 1 and i < n ) then
        out(i) = (tmp(i-1) + 2 * tmp(i) + tmp(i+1))/4
      end if

end subroutine boxCar
```

Race Condition!

Possibly incorrect results

Thread Synchronization

```
call syncthreads()
```

- **Synchronizes all threads in a block**
  - **No thread can pass this barrier until all other threads have reached it**
  - **Necessary to avoid race conditions**
- **Careful when in conditional code!**
  - **Requires all threads reach thread**
- **Does not affect threads in other blocks**
  - **Cannot be used to synchronize between thread blocks**
  - **Only synchronization mechanism between blocks are kernel launches**

Box-Car Filter, Part 3

```
attribute(global) subroutine boxCar(n, in, out)
      real, dimension(*) :: in
      real, dimension(*) :: out
      real, shared, dimension(*) :: tmp
      integer, value :: n, i

      i = (blockidx%x - 1) * blockdim%x + threadidx%x

      tmp[i] = in[i]
      call syncthreads()

      if( i > 1 and i < n ) then
         out(i) = ( tmp(i-1) + 2*tmp(i) + tmp(i+1) )/4
       end if

end subroutine boxCar
```

Determine Maximum in Array

- Multiple threads compete for (write) access to same memory location
  - Other thread updates value we are updating
  - Synchronization within the same block
  - No mechanism if contention from different blocks
- Scan reduction
  - Data parallel approach
  - Can be hard to implement
- Atomic update
  - Guaranteed Cannot be used to synchronize between thread blocks
  - Only synchronization mechanism between blocks are kernel launches

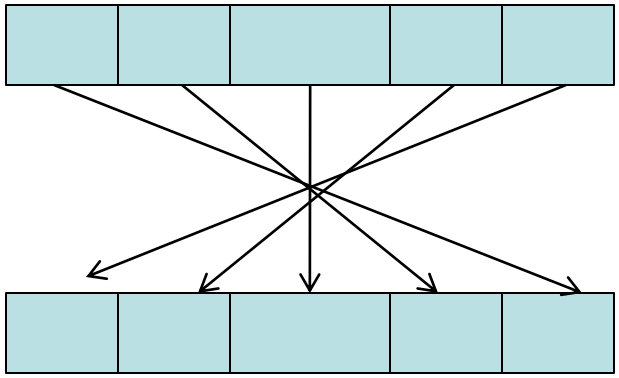
Atomic Memory Operations

- Multiple threads compete for (write) access to same memory location
  - Other thread updates value we are updating
  - Synchronization within the same block
  - No mechanism if contention from different blocks
- Scan reduction
  - Data parallel approach
  - Can be hard to implement
- Atomic update
  - Guaranteed Cannot be used to synchronize between thread blocks
  - Only synchronization mechanism between blocks are kernel launches

Shared memory

- **Problem: Reverse all elements in a (small) array**

```
attribute(global) subroutine reverseKernel(n, a)
     real, dimension(*) :: a
     integer, value :: n
     integer, value :: i

     i = (blockidx%x - 1) * blockdim%x + threadidx%x
     if( i <= n )  a(i) = a(n - i)

end subroutine sqrtKernel
```

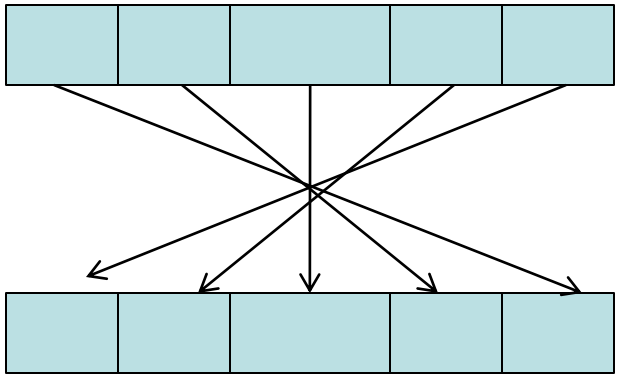


Problem!!
(Data Race)

Thread synchronization and shared memory

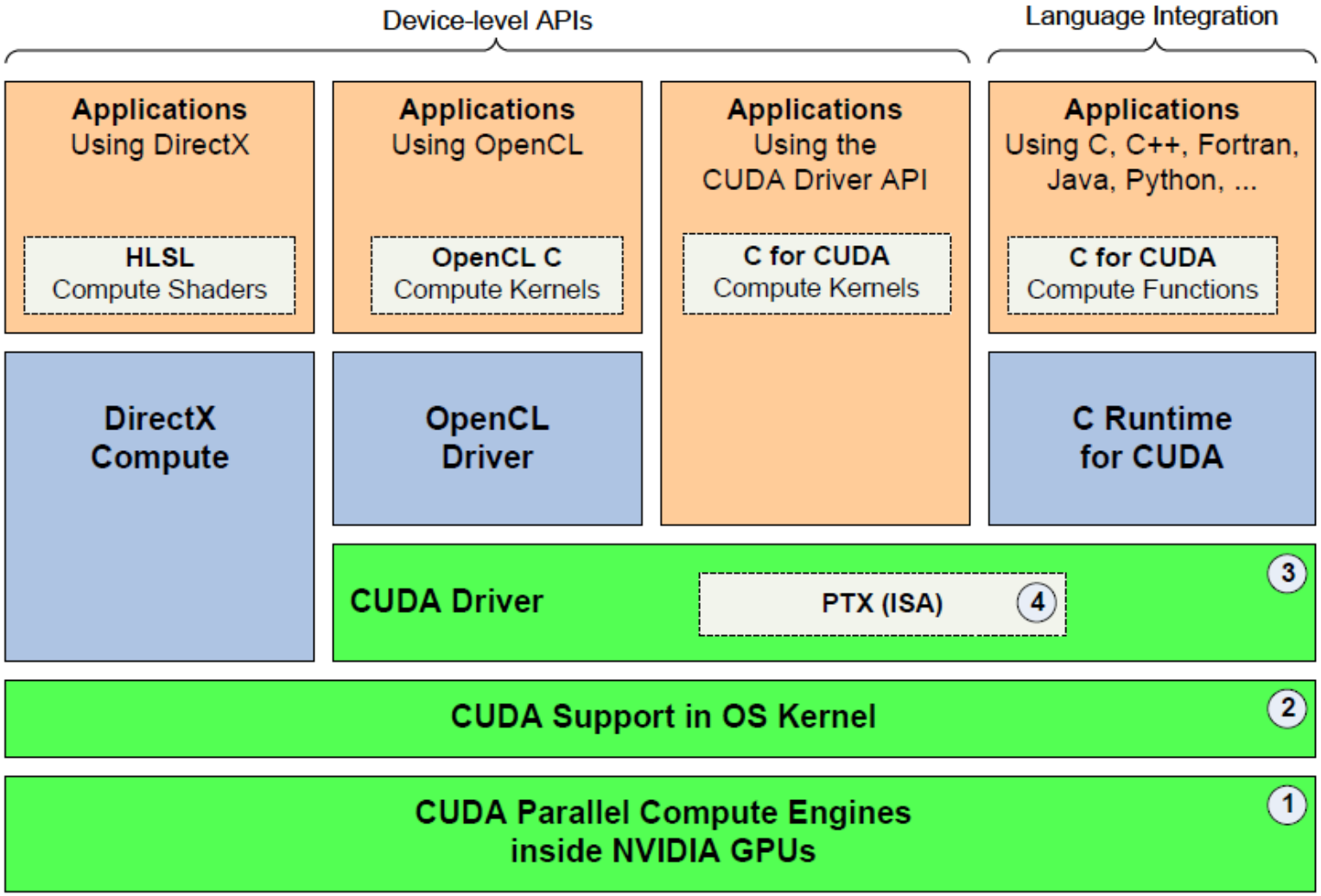
- **Problem: Reverse all elements in a (small) array**

```
attribute(global) subroutine reverseKernel(n, a)
     real, dimension(*) :: a
     integer, value :: n
      real, shared, dimension(*) ::
     integer, value :: i

     i = (blockidx%x – 1) * blockdim%x + threadidx%x
     if( i <= n )  a(i) = a(n - i)

end subroutine sqrtKernel
```



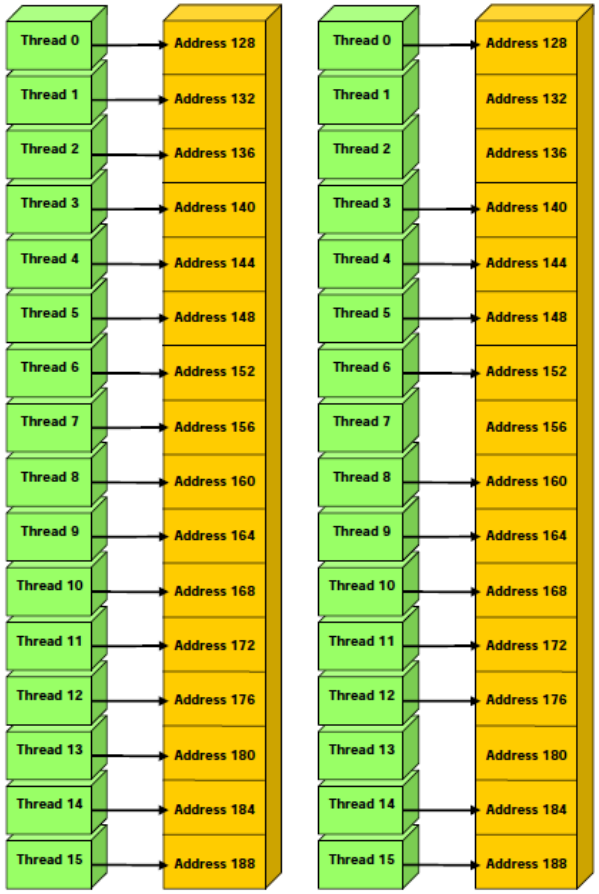
Problem!! (Data Race)

CUDA Software Environment



The Flipside: GPUs need regular memory access

(but newer generation GPUs are getting less picky)

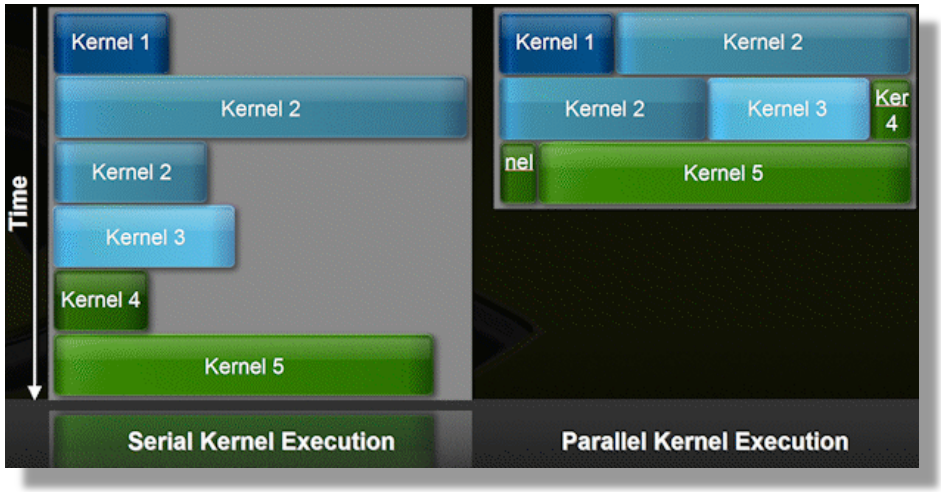


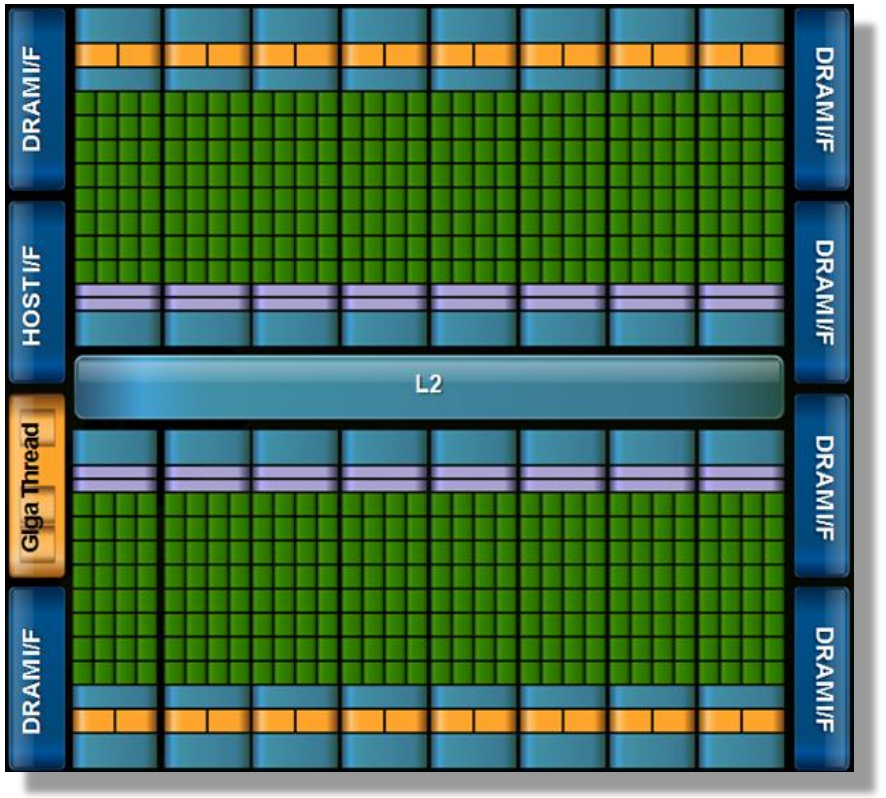
Ideal access pattern

No problem on C1060 and newer

The future: Fermi introduces new level of flexibility

- Multiple kernels executed concurrently
  - Better performance on kernels with low degree of parallelism
- Hardware managed L1, L2 caches
  - Relaxes coalescing requirements
- C++ support on device
- Enhanced atomic performance
- ECC for reliable scaling





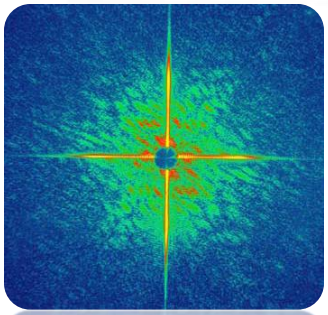
CUDA: Development Environment for NVIDIA GPUs

- Early GPGPU efforts heroic
  - Graphics API (OpenGL, DirectX) no natural fit for scientific computing
- Compute Unified Device Architecture (http://www.nvidia.com/cuda)
  - Supported on all modern NVIDIA GPUs (notebook GPUs, high-end GPUs, mobile devices)
  - Co-Existence with OpenCL (OpenCL basically *IS* CUDA)
- Single Source for CPU and GPU
  - Host code C or C++
  - GPU code C(++) with extensions
    - "Kernel" describes one thread
    - Host invokes a collection of threads
  - nvcc: NVIDIA cuda compiler
- Runtime libraries
  - Data transfer, kernel launch, ..
  - BLAS, FFT libraries
- Simplified GPU development, but still "close to the metal"!
- NEXUS: Visual Studio plug-in for GPU development

IDL users/Data analysts (would like to) devote their time to more important things than developing (low level) code

GPULib's Goal: Acceleration without the pain

GPULib:
High-Productivity GPU Computing



- IDL (ITT Vis), MATLAB (Mathworks)
  C, Fortran
- Rich set of data parallel kernels
- Extensible with proprietary kernels
- Seamless integration into host language
- Explicit or implicit management of address spaces
- Interface to Tech-X' FastDL for multi-GPU/distributed memory processing

http://gpulib.txcorp.com
(free for non-commercial use)

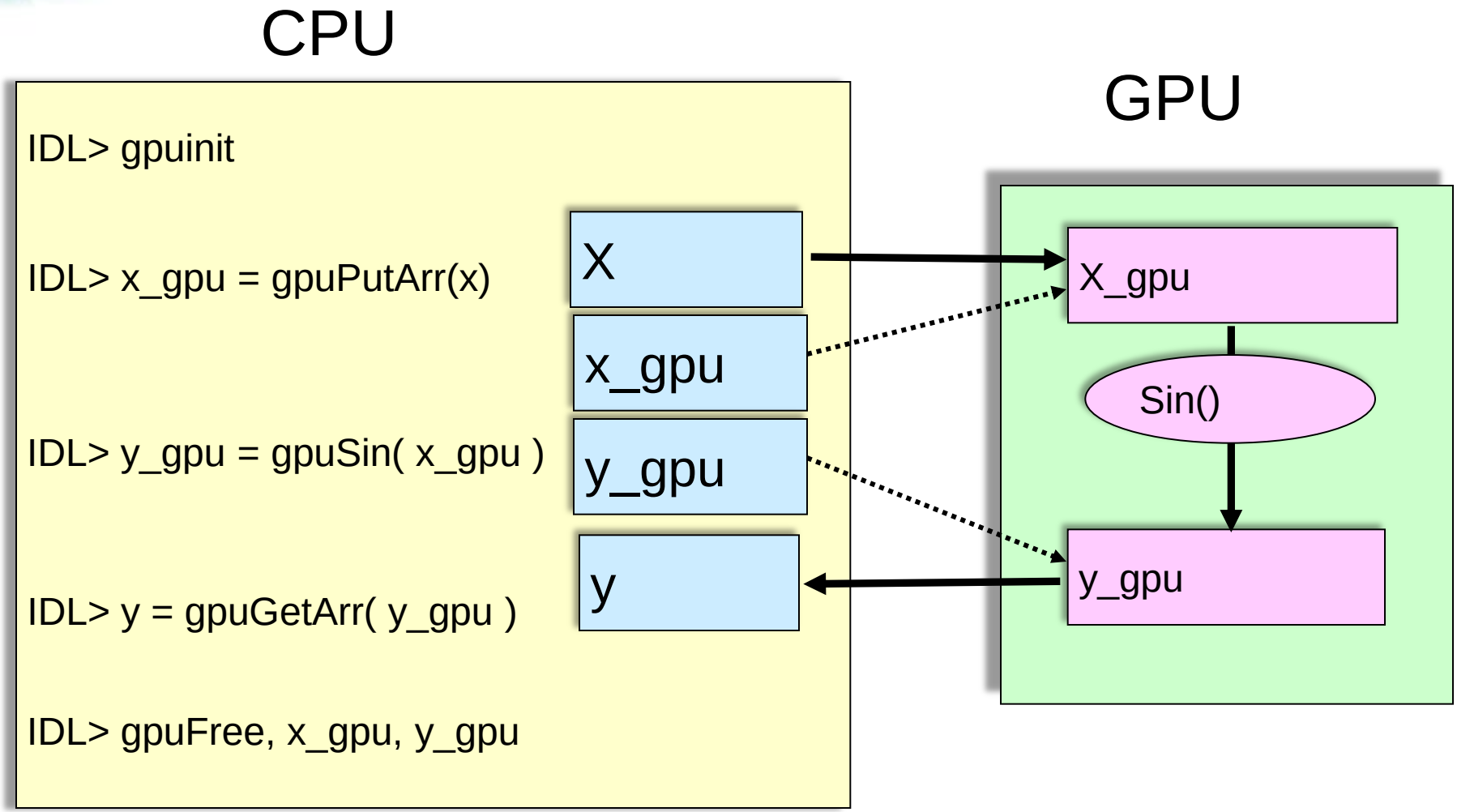
FastDL

Messmer, Mullowney, Granger, *"GPULib: GPU computing in High-Level Languages",* Computers in Science and Engineering, 10(5), 80, 2008.

GPULib enables development of complex algorithms on GPUs

- Data objects on GPU represented as structure/object on CPU
  - Contains size information, dimensionality and pointer to GPU memory
- GPULib provides a large set of vector operations
  - Data transfer GPU/CPU, memory management
  - Arithmetic, transcendental, logical functions
  - Support for different types (float, double, complex, dcomplex)
  - Data parallel primitives, reduction, masking (total, where)
  - Array operations (reshaping, interpolation, range selection, type casting)
  - NVIDIA's cuBLAS, cuFFT
- Extensible architecture
  - Customized kernels

An example of using GPULib in IDL



Just add ‘gpu’ prefix to IDL commands ...

- Memory allocation on GPU

```
y_gpu = gpuFltarr(100, 100)
```

- Data transfer

```
x_gpu = gpuPutArr(x)
```

- Binary operators both plain and affine transform

```
z_gpu = gpuAdd(x_gpu, y_gpu)
z_gpu = gpuExp(a, b, x_gpu, c, d)
```

- IDL intrinsics

```
gpuInterpolate, gpuTotal, gpuCongrid, gpuSubArr,
gpuReform, gpuWhere, gpuRandomu, gpuFltarr, gpuComplex,
gpuMatrix_multiply …
```

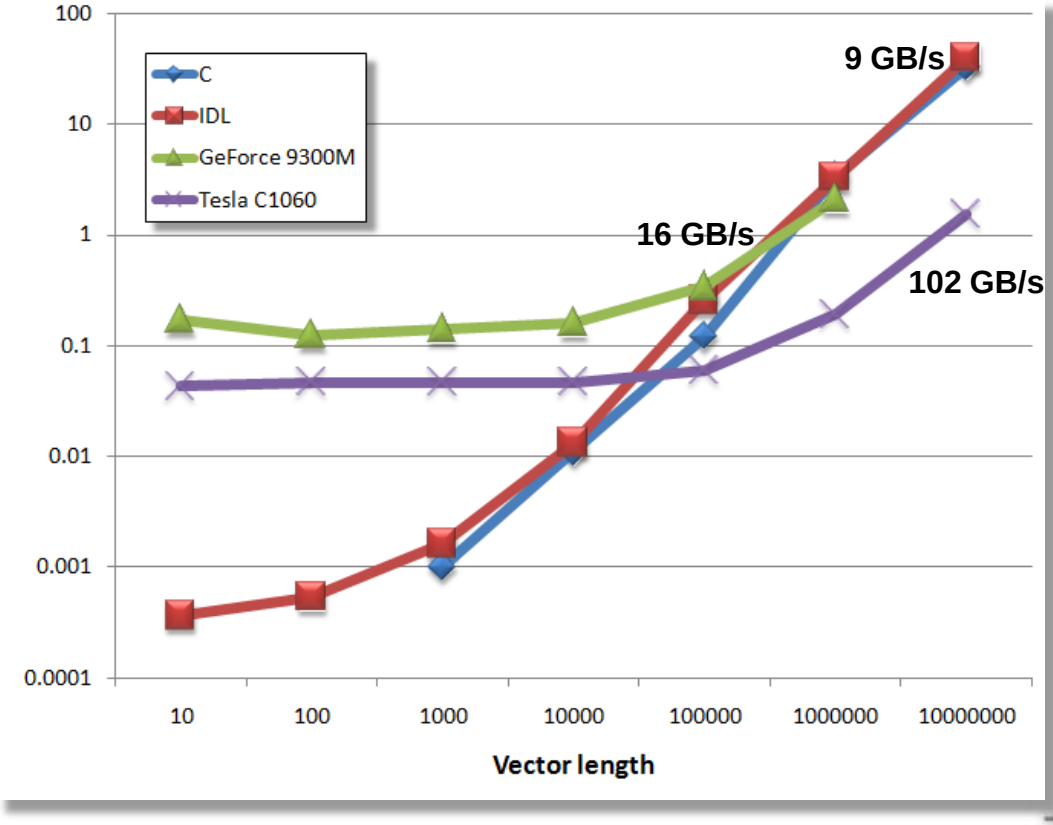
- IDL structure contains all information about GPU object

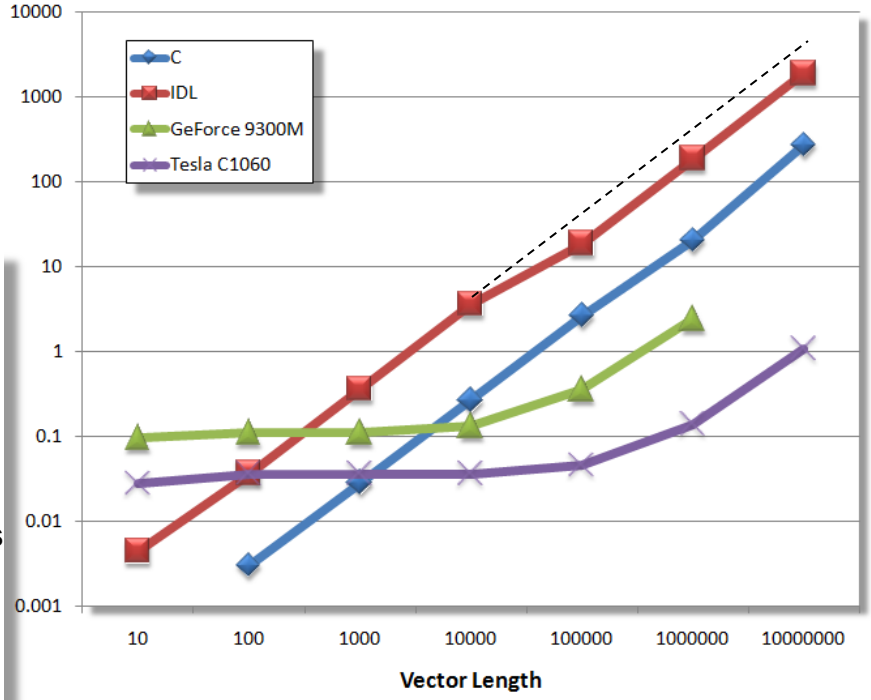
```
type, n_elements, n_dimensions, dimensions, handle
```

- Major changes with IDL 8

Comparison of different vector implementations

$z = x + y$

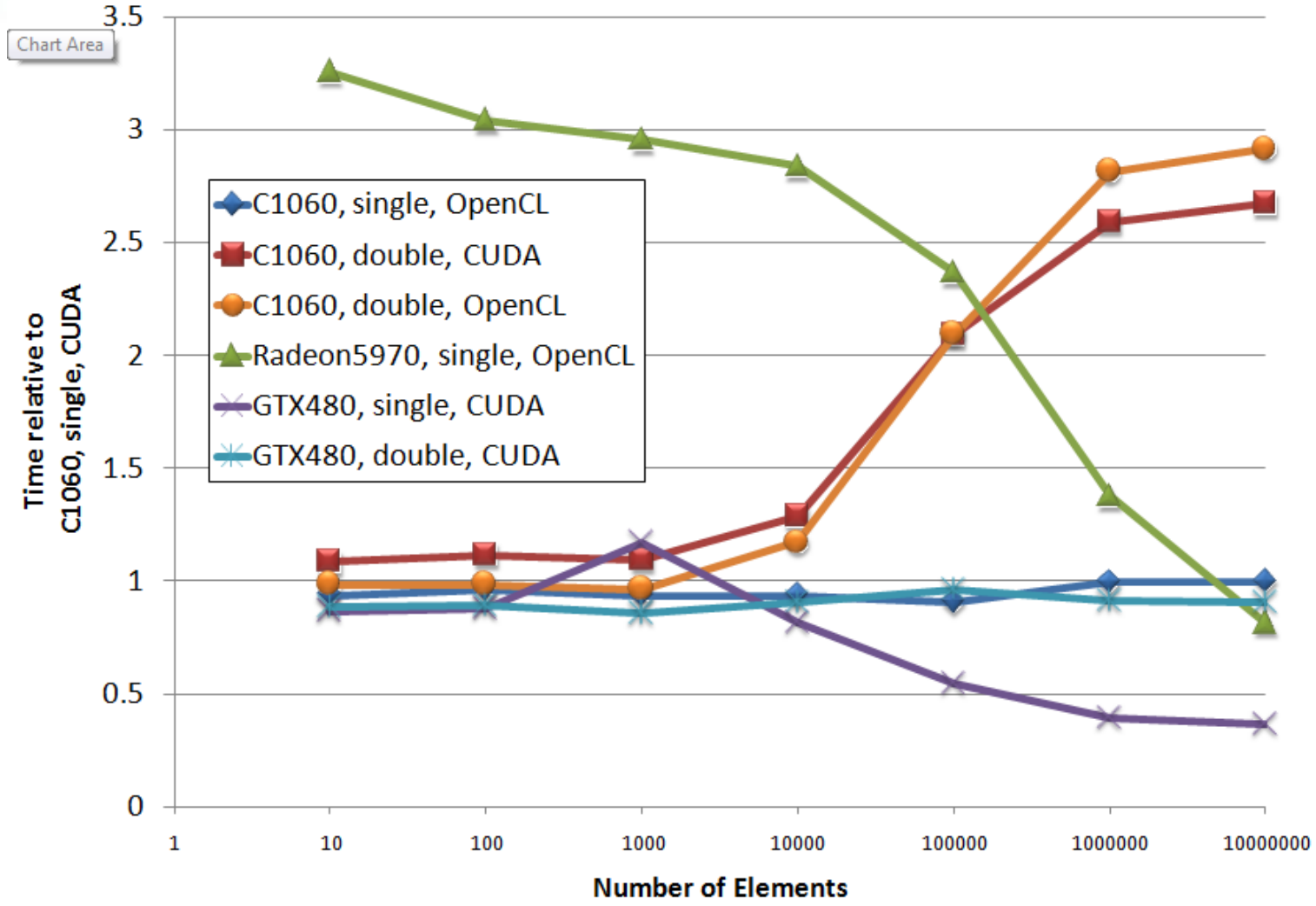




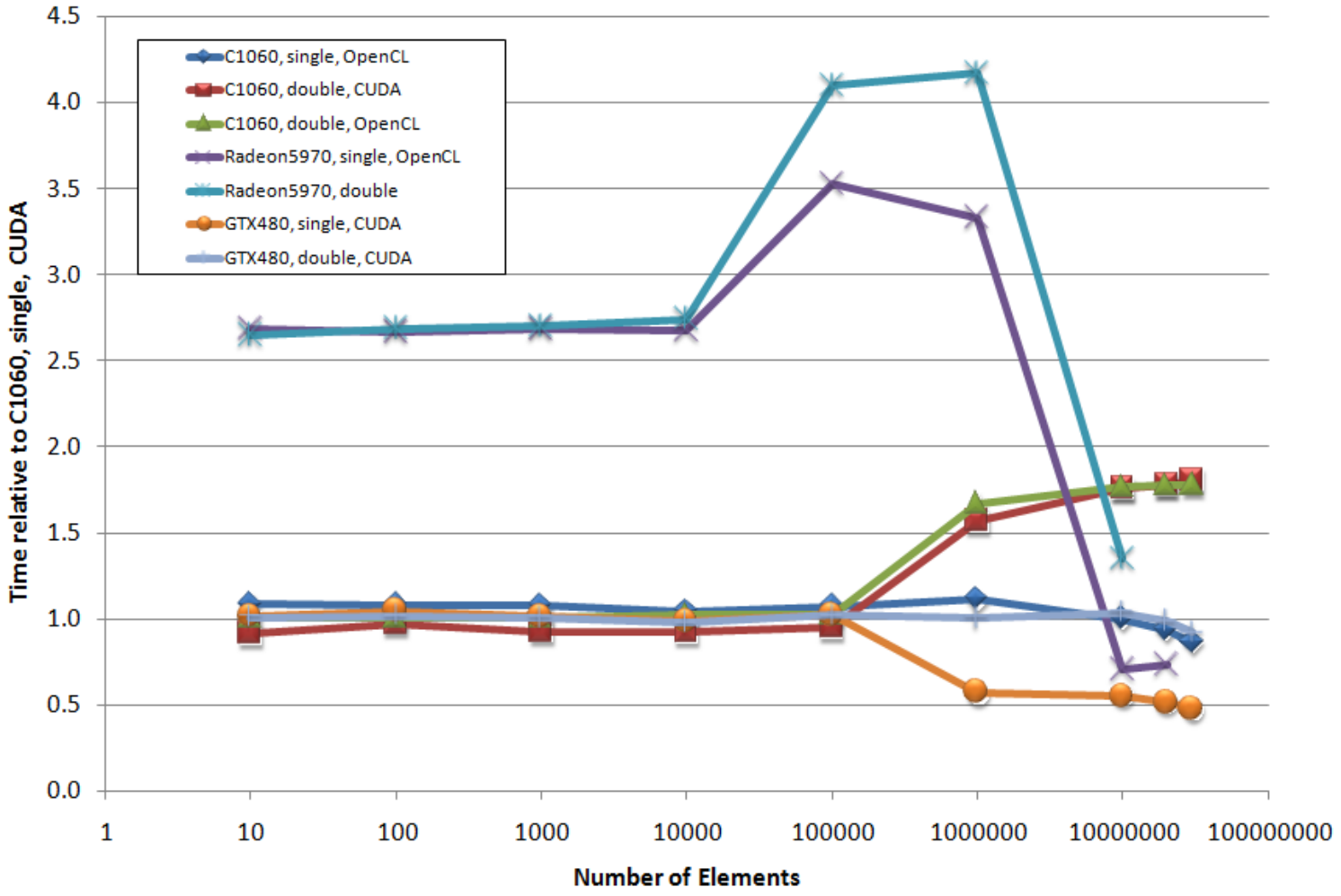
$z = a\ exp(b\ x + c) + d$

GeForce 9300M GS: 16 cores, 9GB/s
Tesla C1060: 240 cores, 102GB/s
CPU: 2.4GHz Core2Duo E4600

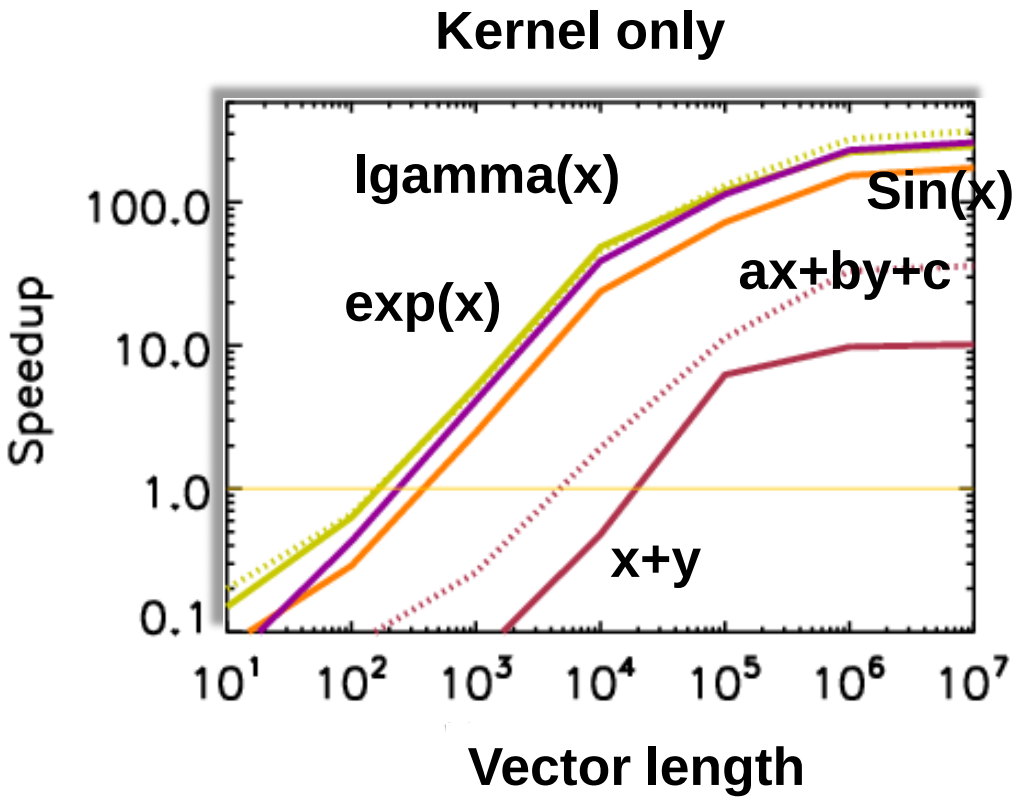
Kernel Execution Time on Compute Bound Problem, y = exp(x)

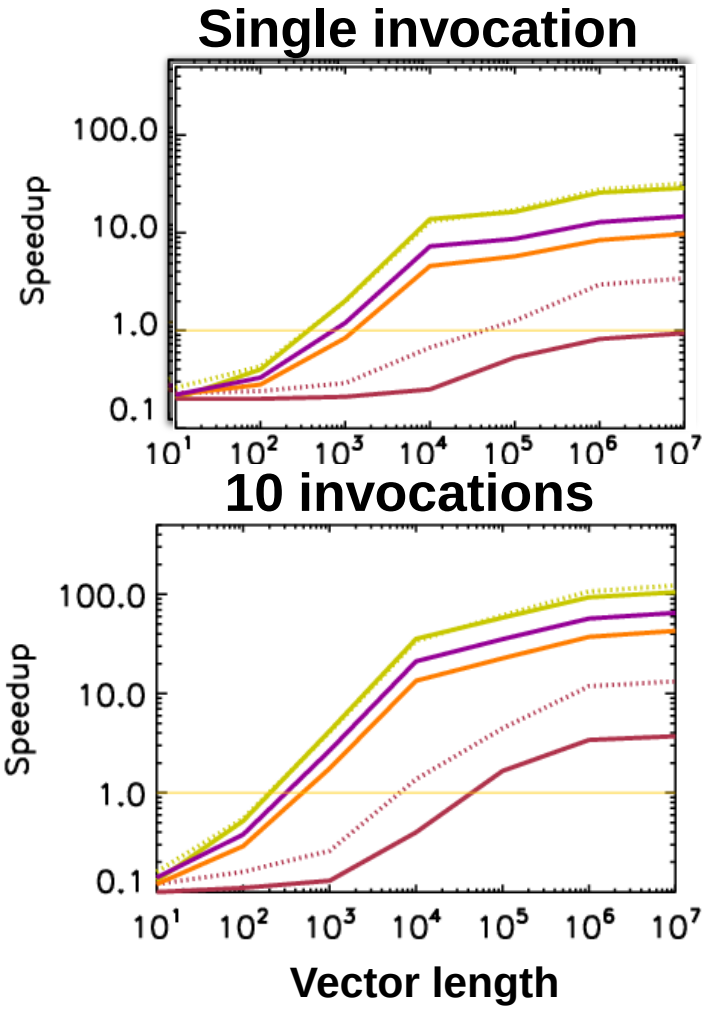


Kernel Execution Time on Memory Bandwidth Limited Problem, z = x + y



Key to performance: Multiple kernel invocations per CPU-GPU transfer





GPULib offers different interfaces for different needs

- Plain IDL:

```
rho = Sqrt(x * x + y * y)
```

144.8s

- Procedural interface with explicit memory management:

```
tmp1 = gpuMake_array(…)
tmp2 = gpuMake_array(…)
rho = gpuMake_array(…)
gpuMult, x, x, tmp1
gpuMult, y, y, tmp2
gpuAdd, tmp1, tmp2, tmp1
gpuSqrt, tmp1, rho
gpuFree, tmp1
gpuFree, tmp2
```

7.2s

- Functional interface with simplified memory management

```
rho = gpuSqrt(gpuAdd(gpuMult(x, x), $
                     gpuMult(y, y))
```

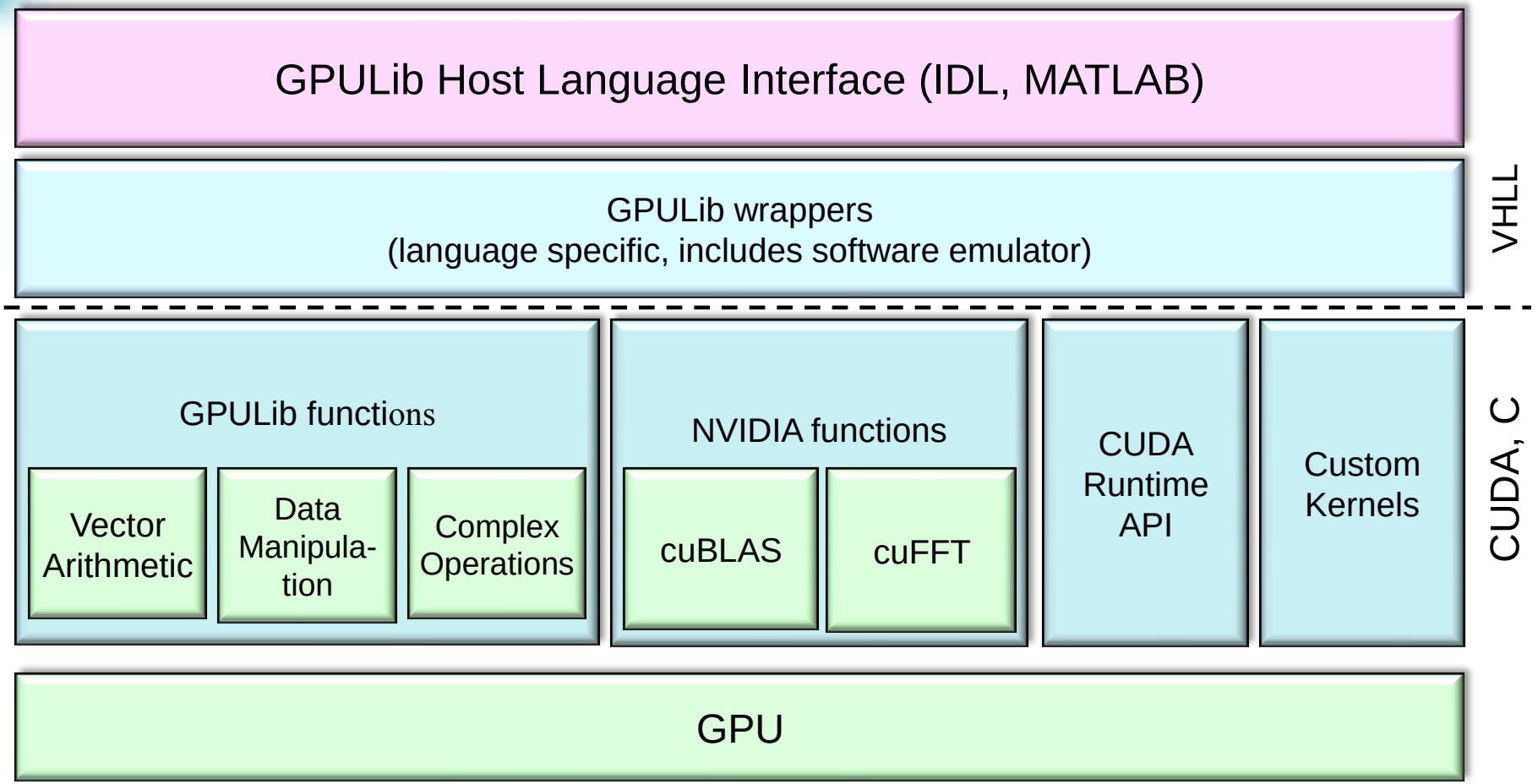
15.3s

- Since IDL 8.0: Operator overloading

```
rho = gpuSqrt(x * x + y * y)
```

15.4s

GPULib's Extensible Architecture



Tech-X GPU Consulting

- Tech-X supports clients throughout entire project life-cycle
  - Problem definition
    - Performance analysis
    - Algorithm Review
  - Implementation and optimization
    - Parallelization (MPI, OpenMP)
    - GPULib implementation
    - CUDA implementation
  - Data analysis and knowledge discovery
    - Large-Volume data analysis
    - High-end and autostereoscopic visualization
    - Workflow optimization

- GPU consulting clients involve Industry, Government and Academia
  - Computational finance
  - Ray-tracing
  - Optimization of constraint solver
  - Particle-based solvers
  - Computational fluid dynamics
  - ...

A typical Consulting Example: GPU acceleration of MATLAB Ray-Tracing Code

- Performance analysis revealed bottleneck in solution of small matrices
- Tech-X' mathematicians discovered that linear system was reducible to explicit form
  - Order of magnitude speedup by algorithmic changes
- Rapid prototyping of GPU acceleration via GPULib
  - Demonstration of benefit of GPUs
- Implementation of dedicated CUDA kernel
  - Ultimate performance
- Overall speedup ~ 1000x compared to original implementation

=> Result of comprehensive analysis, rather than just code translation

  - Tech-X scientific staff with broad domain expertise a huge asset

Measure, Think, Code .... (Repeat)

Amdahl's Law, the key to successful optimization

P = Parallelizable part of the code
(1-P) = Serial part of the code

Parallel speedup $S(N) = \frac{T_1}{T_N} = \frac{1}{(1-P) + \frac{P}{N}}$ (Amdahl)

Maximum achievable speedup: $S(N \to \infty) = \frac{1}{1-P}$

- use 'optimizable' part
  O = Part of code that benefits from optimization
  (1-O) = Part that does not benefit from optimization

  Speedup achieved by optimization by factor α: $S(\alpha) = \frac{1}{(1-O) + \frac{O}{\alpha}}$

  Maximum achievable speedup $S(\alpha \to \infty) = \frac{1}{1-O}$

$\Rightarrow$ In order to get a significant speedup, O has to be a very large fraction of the overall time

Examples

Problems solved by GPULib and Tech-X Custom Kernels

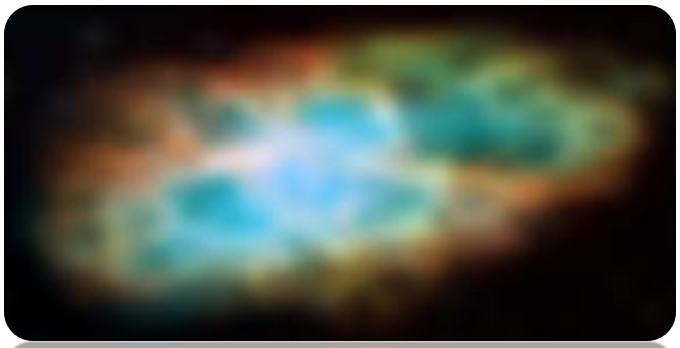
Image Deconvolution

- Image is convolved with detector point-spread function:

$$I_{obs}(x,y) = \int I_{true}(x-u, y-v)P(u,v)dudv$$

- Clean image by (complex) division in Fourier space:

$$I_{true}(x,y) = FFT^{-1}(FFT(I_{obs}) / FFT(P))$$

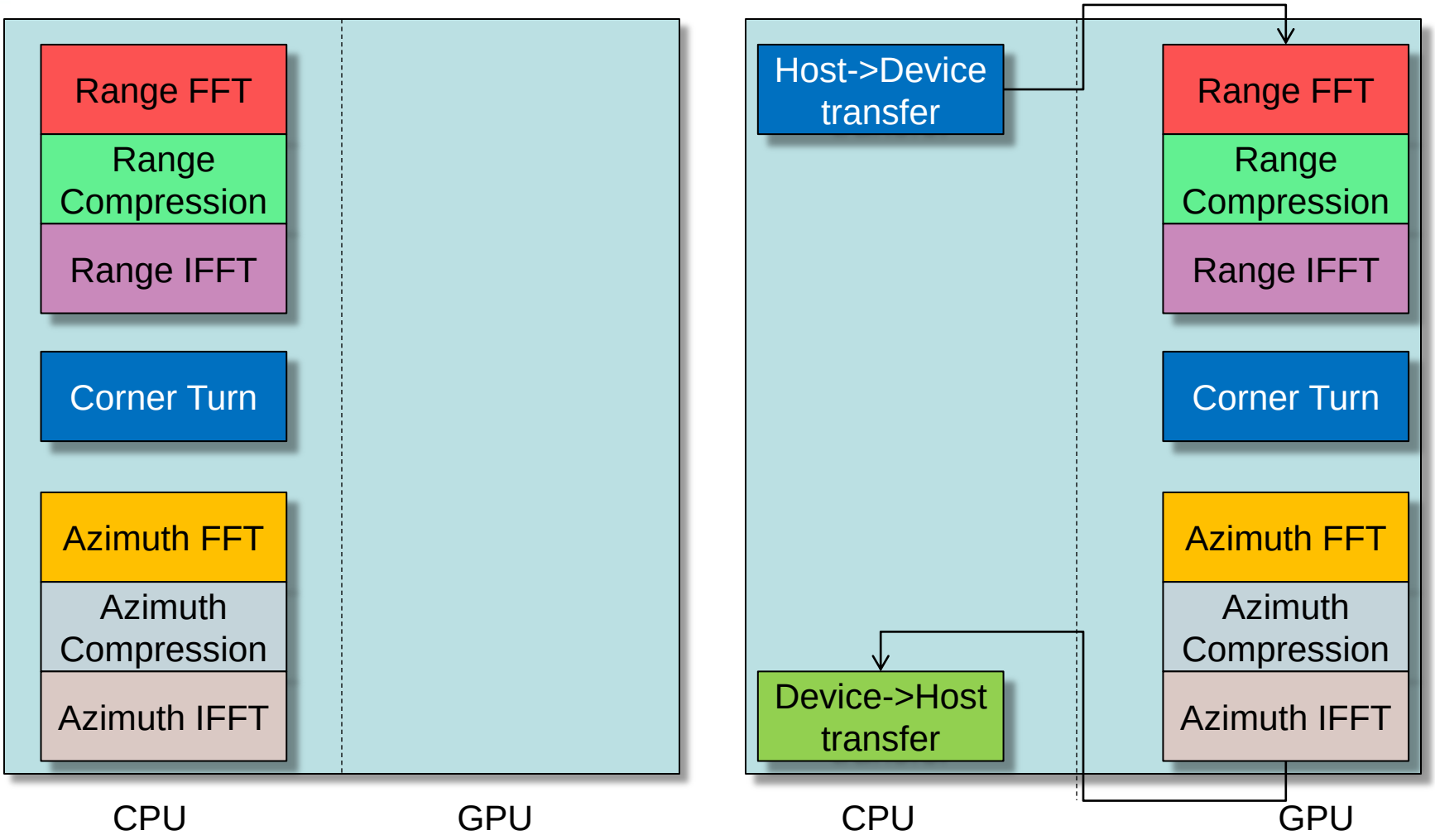


- Speedup ranging from 5x – 28x
  for 256x256 – 3kx3k images

```
gpu_img = gpuFFT(img)
gpu_img = gpuDiv(gpu_img, gpu_psf)
gpu_clean = gpuFFT(gpu_img_fft, /INVERSE)
clean = gpuGetArr(gpu_clean)
```

Range-Azimuth Processor

A GPULib-based Range-Azimuth Processor



A GPULib-based Range-Azimuth Processor

```
gpuinit
```

```
d = gpuMallocHost(nsample * npulse,  6L)
d[0:*] = data
x_gpu = gpuComplexarr(nsample, npulse)
```

Data Allocation

```
g_func1= gpuPutArr(func1)
gpuPutArr, d, x_gpu
x_gpu.n_dimensions = 2
x_gpu.dimensions = [nsample, npulse]
```

Data transfer

```
x_gpu = gpuFFt(x_gpu, LHS=x_gpu, /batch,
    plan=plan_forward, /destroy)
```

```
for i=0,npulse-1 do begin
    gpuView, x_gpu, nsample*i, nsample,
           x_gpu_view
    gpuMult, x_gpu_view, g_func1,
           x_gpu_view, error=err
end
```

Range Compression

```
x_gpu = gpuFFt(x_gpu, LHS=x_gpu, /batch,
    /INVERSE, /destroy)
```

```
y_gpu = gputranspose(x_gpu)
```

```
y_gpu = gpuFFt(y_gpu, LHS=y_gpu, /batch,
    plan=plan_forward, /destroy)
```

```
for i=0,nsample-1 do begin
    gpuView, y_gpu, npulse*i, npulse,
           y_gpu_view
    gpuMult, y_gpu_view, g_func2,
           y_gpu_view, error=err
end
```

Azimuth Compression

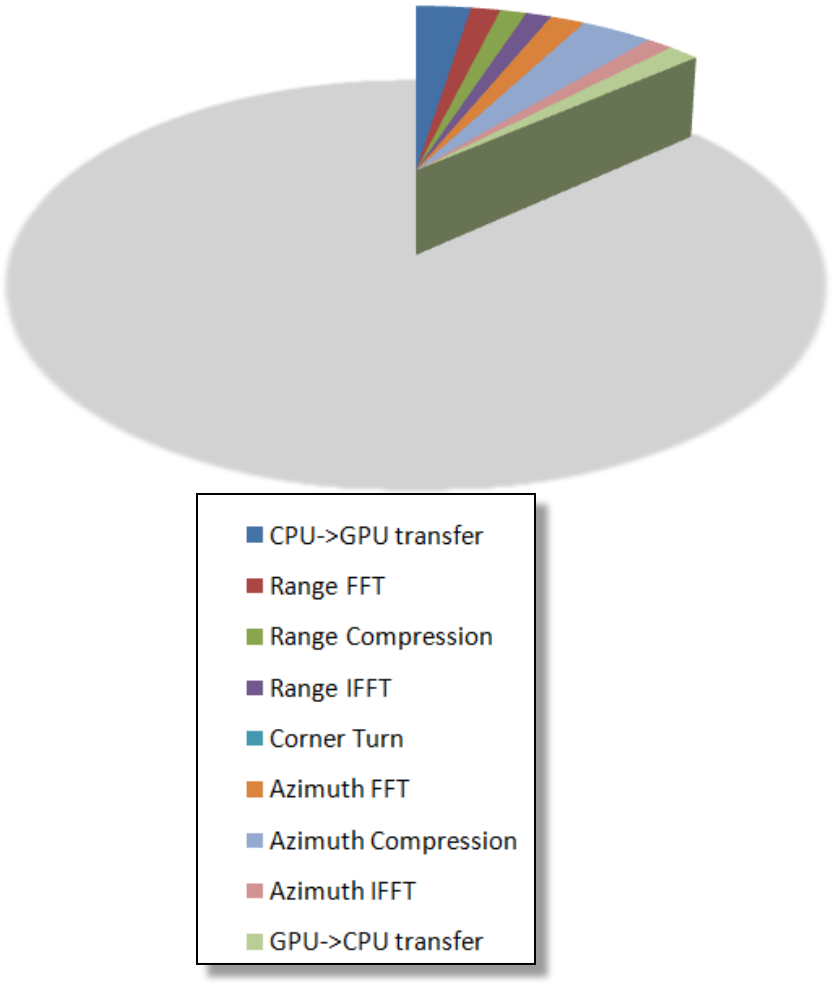
```
y_gpu = gpuFFt(y_gpu, LHS=y_gpu, /batch,
    /INVERSE, plan=plan_inverse, /destroy)
```

```
gpugetarr, y_gpu, d
```

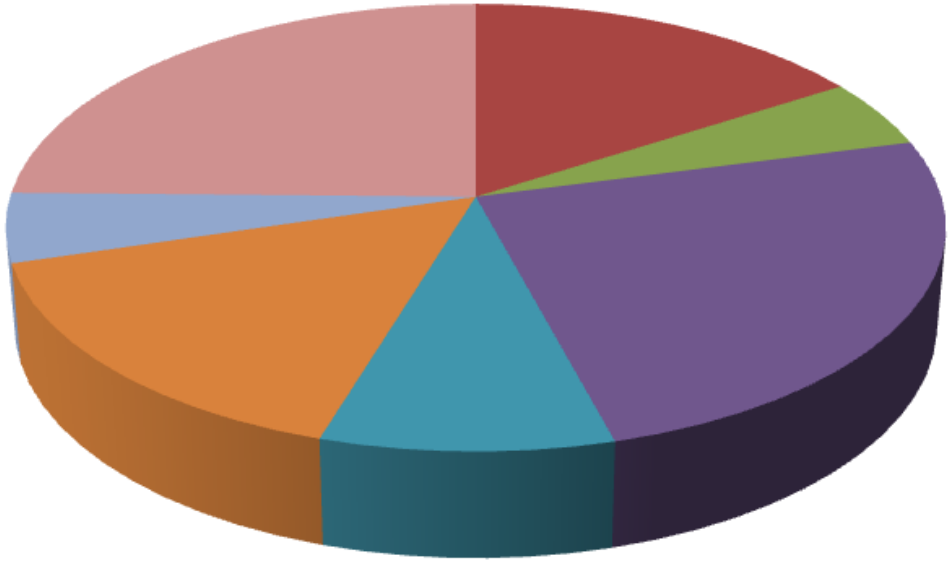
GPULib acceleration of Range-Azimuth processor

CPU+GPU
(2.4 GHz Core2Duo E4600 + Tesla C1060)

20000x6500 range-azimuth dataset
(single precision complex)



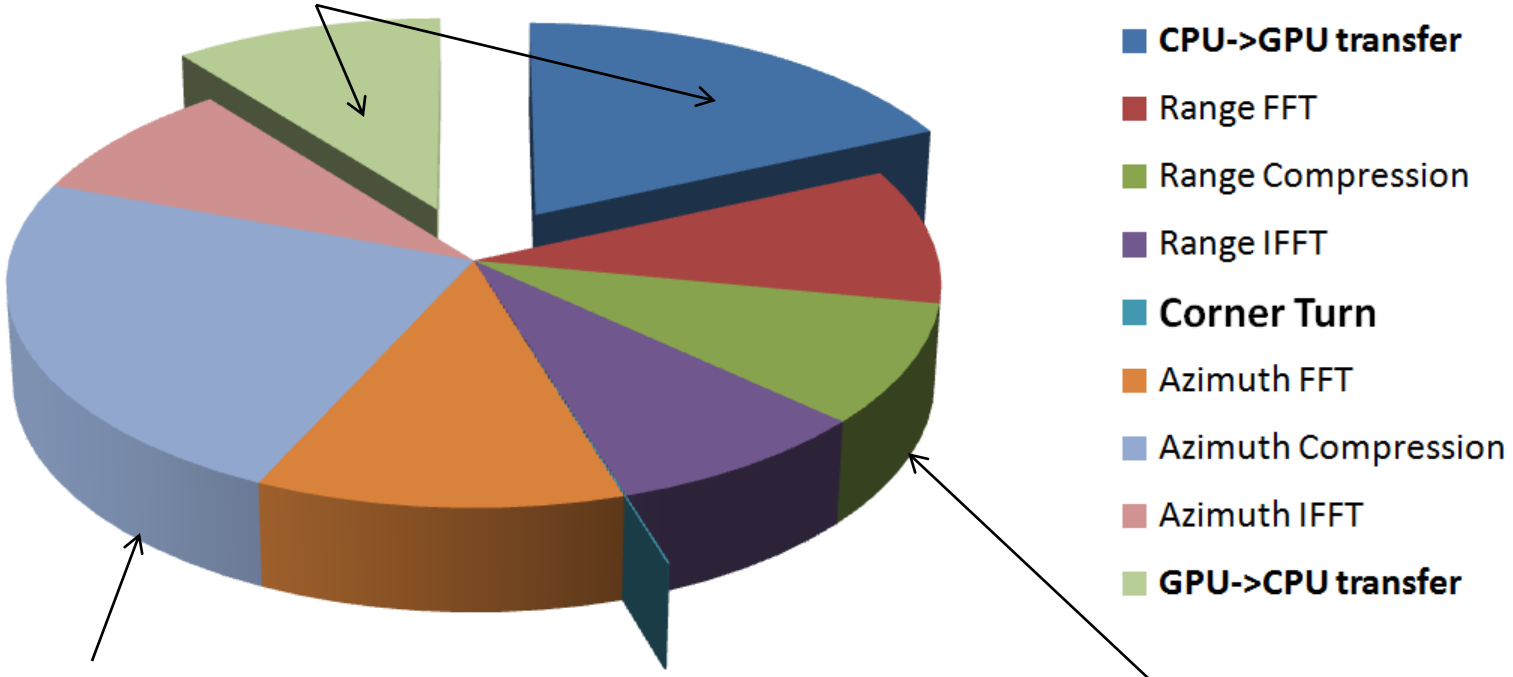
CPU
(2.4 GHz Core2Duo E4600)



Detailed profile of GPULib accelerated range-azimuth processor reveals room for improvement

Data Transfer
Problem: Currently using blocking transfer
Solution: Asynchronous transfer for latency hiding

26880x4912 range-azimuth dataset



Azimuth Compression
Problem: Large number of kernel invocations on fairly short vectors
Solution: Specialized kernel

Range Compression
Problem: Smaller number of kernel invocations on relatively long vectors
Solution: Specialized kernel

Spectral Angle Mapper

A GPULib-based Range-Azimuth Processor

Dot products

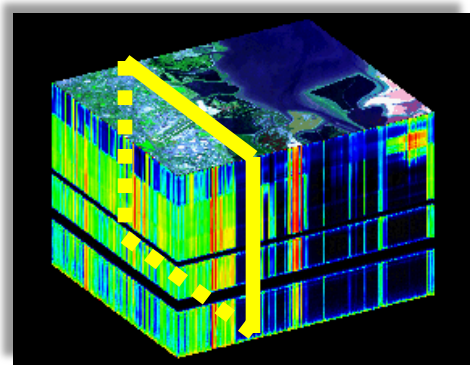
```
For i=0, nlines – 1 do begin
  res[*, *, i] =
    matrix_multiply(ref, a[*,*,i], /a)
end
```

Channel norm

```
a = reform(a, nbands, nsample*nlines)
r = sqrt(total(a^2, 1))
r = reform(r, nsamples * nlines)
```

Normalization

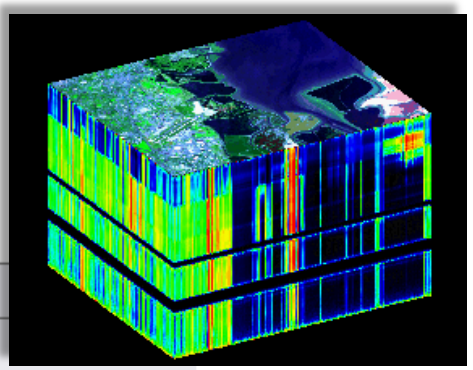
```
res = reform(res, nref, nsamples *nlines)
res = transpose(res)
for i = 0, nref do
    res [ *, i] /= r
```



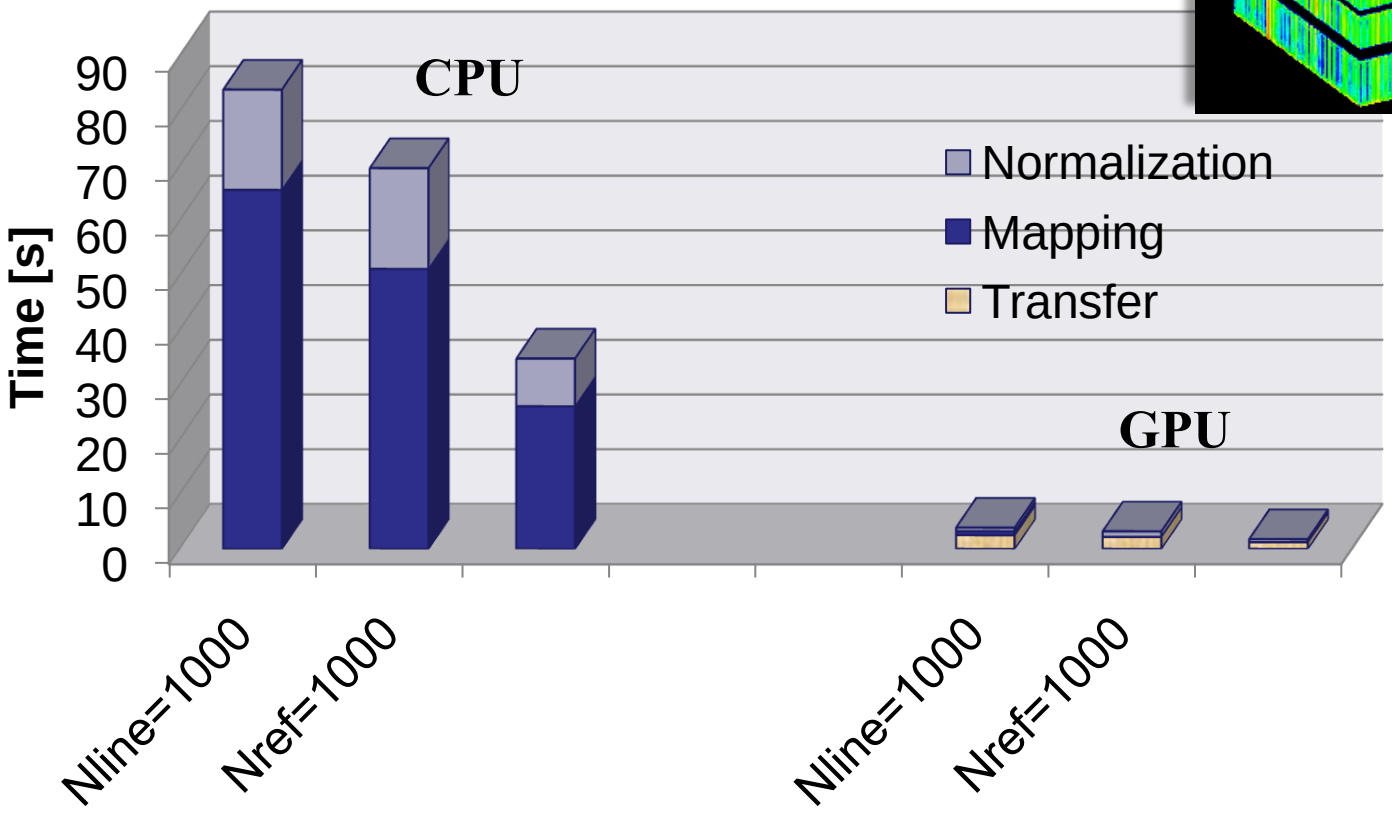
*

Ref. spectra

Spectral Angle Mapping on GPU



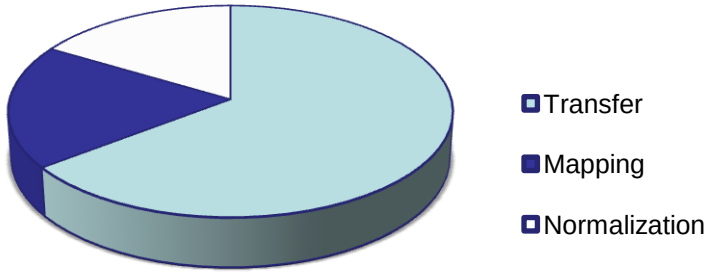
224 Channels, 614 Samples, 512 Lines,
500 reference spectra



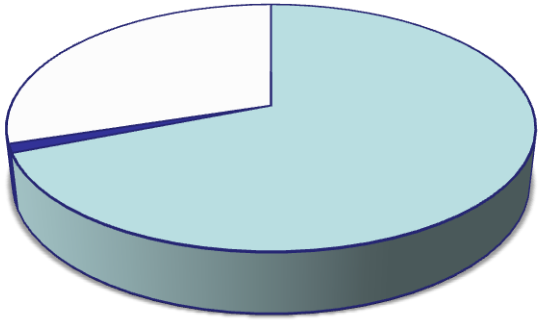
GPU SAM dominated by data transfer cost

- SAM significantly accelerated compared to CPU
  - > 20x speedup overall
- Mapping a single kernel
  - SGEMM
- Normalization a collection of kernels
  - Norm, sqrt, division, ...
- Data transfer significant part of cost
  - Increase to 3 GB/s from 1.5 GB/s by using page-locked memory
  - Asynchronous data transfers to the rescues
- Consider entire application, not just isolated kernels!
  - Entire flow from disk -> host memory -> GPU and back

224 Channels, 614 Samples, 1000 Lines, 500 reference spectra



224 Channels, 614 Samples, 512 Lines, 500 reference spectra



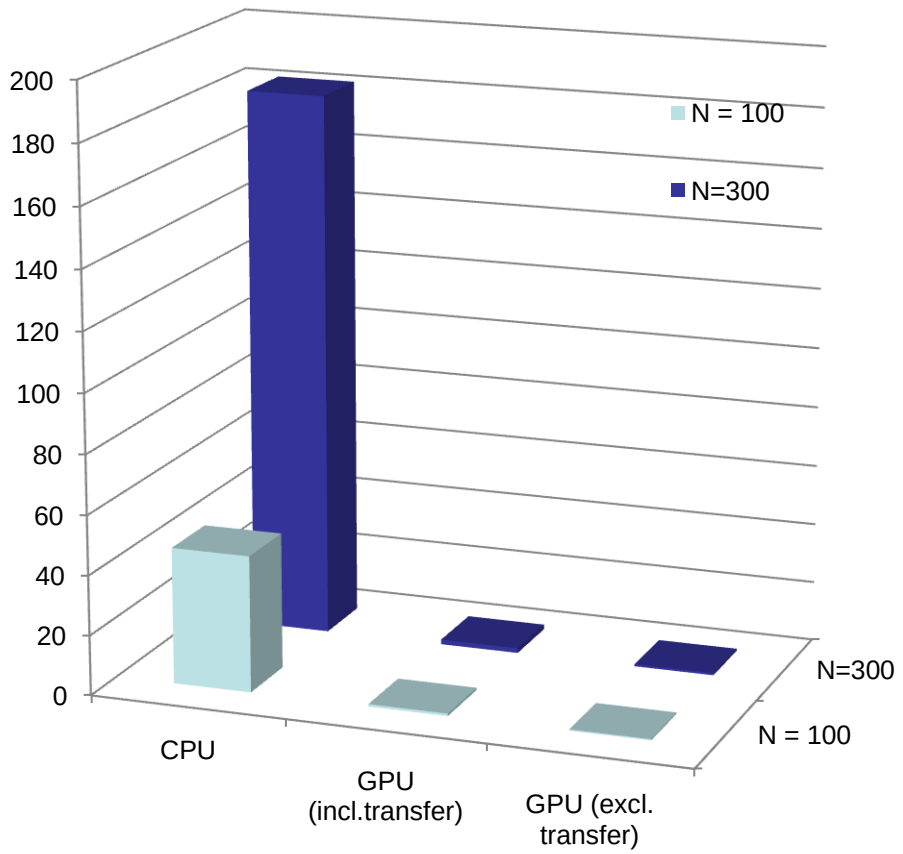
Implicit Solver

GPU acceleration of Option Pricing
(solving lots of small 1D diffusion equations)

- Large ensemble of 1D diffusion equations
- Tri- or pentadiagnoal matrices (sparse!)
- 360,000 systems with 100 – 300 unknowns

- Reference implementation in MATLAB

- Tech-X Custom kernel
  N=300: 182s CPU -> 1.6s GPU
  N=100: 45s CPU -> 0.7s GPU



A GPULib-based Conjugate-Gradient Solver

Initialization

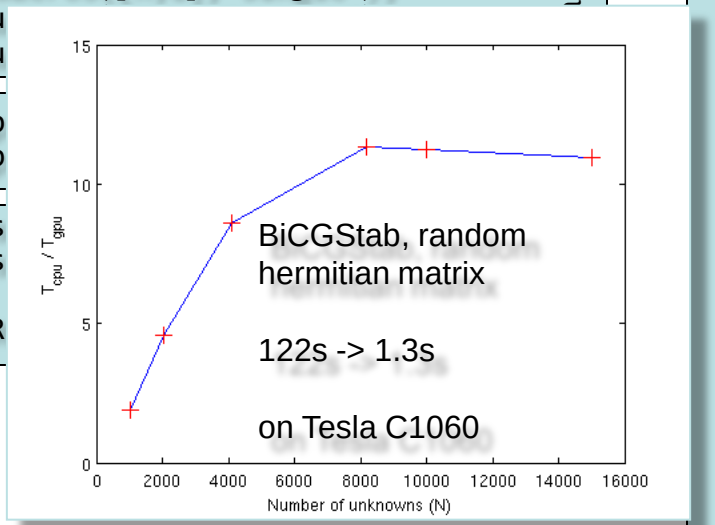
```
A = single(A);
b = single(b);

[n, m] = size(A);

gpuInit()
```

Allocation

```
aGpu = gpuZeros([n,m],'single');
bGpu = gpuZeros([n,1],'single');
rGpu = gpuZeros([n,1],'single');
pGpu = gpuZeros([n,1],'single');
qGpu = gpu
xGpu = gpu
```

```
gpuSet(aGp
gpuSet(pGp
```

```
normB  = s
normR  = s

nu = normR
```

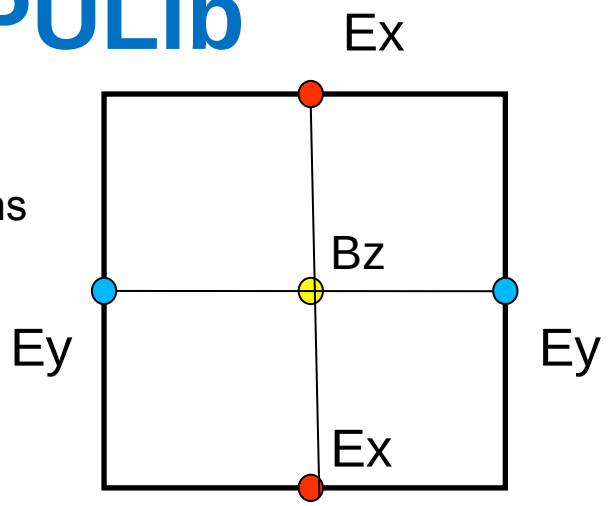


Main iterations

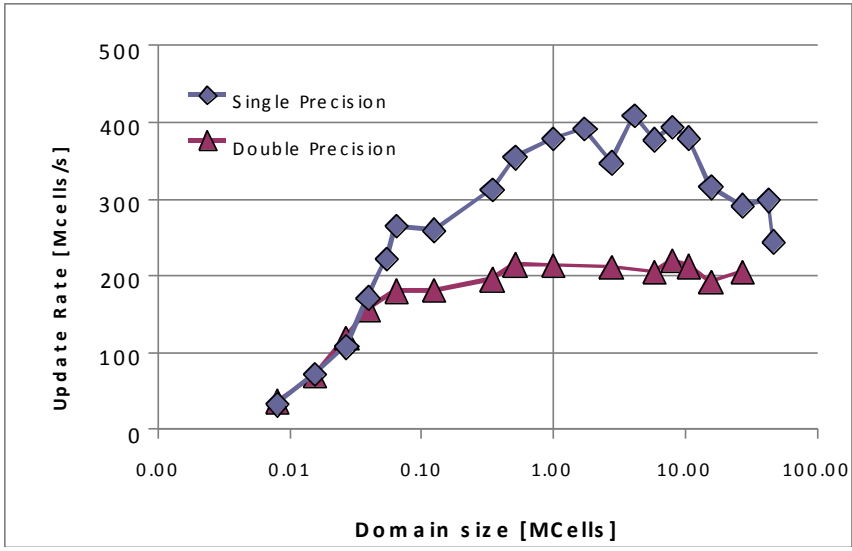
```
for k=1:maxiter

    relErr = normR/normB;

    if(relErr < tol)
           x = gpuGet(xGpu);
           return;
    end

    qGpu   = gpuMatVecMultiply(aGpu,pGpu);

    mu     = gpuDot(pGpu, qGpu);

    alpha  = nu / mu;

    xGpu   = gpuPlusAT(single(1.0), xGpu,
    alpha, pGpu, single(0.0));

    rGpu   = gpuPlusAT(single(1.0), rGpu, -
    alpha, qGpu, single(0.0));

    nuP    = gpuDot(rGpu,rGpu);
    beta   = nuP / nu;

    pGpu   = gpuPlusAT(single(1.0),rGpu,
    beta, pGpu, single(0.0));

    nu    = nuP;

    normR  = sqrt(nuP);

end
```

Accelerating Memory Bandwidth limited applications

3D FDTD in GPULib



- Widely used for solving time-dependent Maxwell's equations
- $$B_z^{n+1} = B_z^n + dt\left(\frac{E_x^{y+1} - E_x^{y+1}}{dy} - \frac{E_y^{x+1} - E_y^{x+1}}{dx}\right)$$
- 3D FDTD
  - Cut-Cell or stair-stepped boundaries
  - Uses VORPAL geometry output
- Performance
  - Up to 400 Mcells/s on
    - ~70% theoretical memory bandwidth
  - ~10 Mcells/s on CPU



⇒ ~ 40-50x speedup compared to CPU based implementation
  - Comparable to ~48 Franklin cores

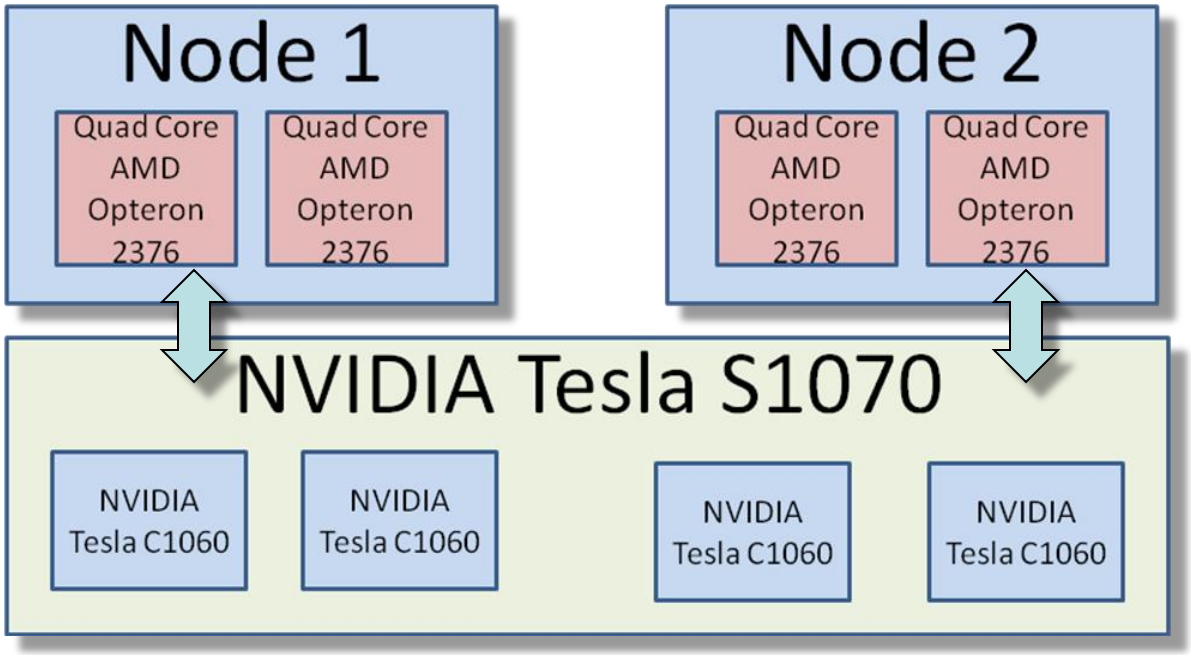
- Double precision hit only due to memory bandwidth
  - Think about your units!
- Multi-GPU via message passing among GPU accelerated nodes
  - Eg. 2.6x speedup on a 3 GPU 'cluster' (PSC)
- Now part of VORPAL (ongoing)

Out-of-core LU factorization

(An example of GPU cluster computation)

Oxygen: Tech-X' Production cluster with GPU accelerated nodes

- 32 nodes, each with dual quad core Opteron
- 8GB RAM per node
- Infiniband interconnect
- Lustre file system
- 4 nodes accelerated with 2 Tesla GPU blades



Benefit of GPUs not limited to FDTD: Boundary Element Methods

– Large systems of linear equations
  • O(100k)-O(1M) unknowns or larger

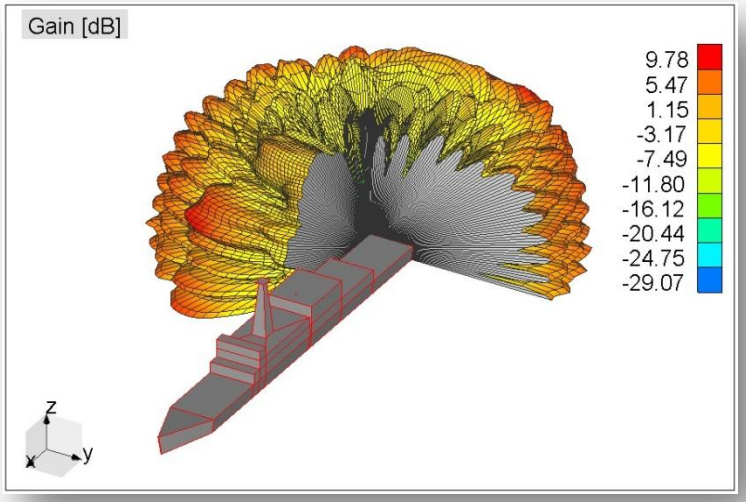
- Large matrices
  - 100k unknowns: 80 GB
  - 1M unknowns: 8 TB

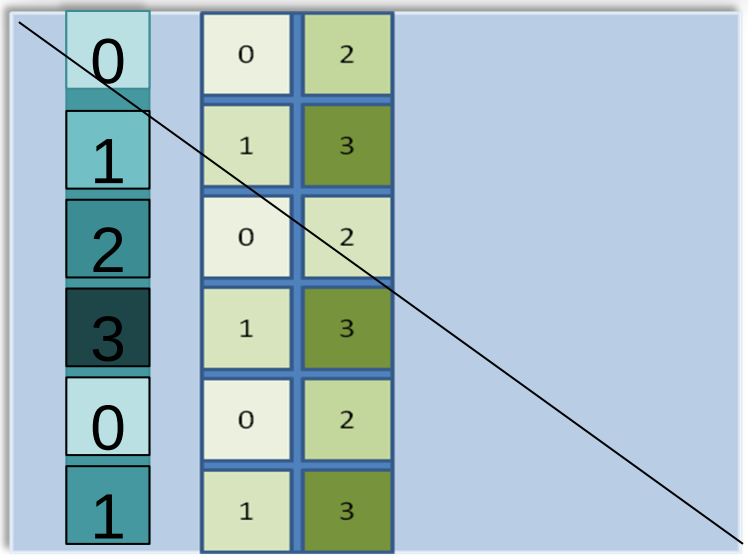
- Direct or iterative method
  - LU for dense systems, reuse for different RHS
  - Iterative possibly faster for single solution

- Too large to fit into memory of small cluster

- Solution time scales with $O(N^3)$
  - Interesting problems take days to weeks

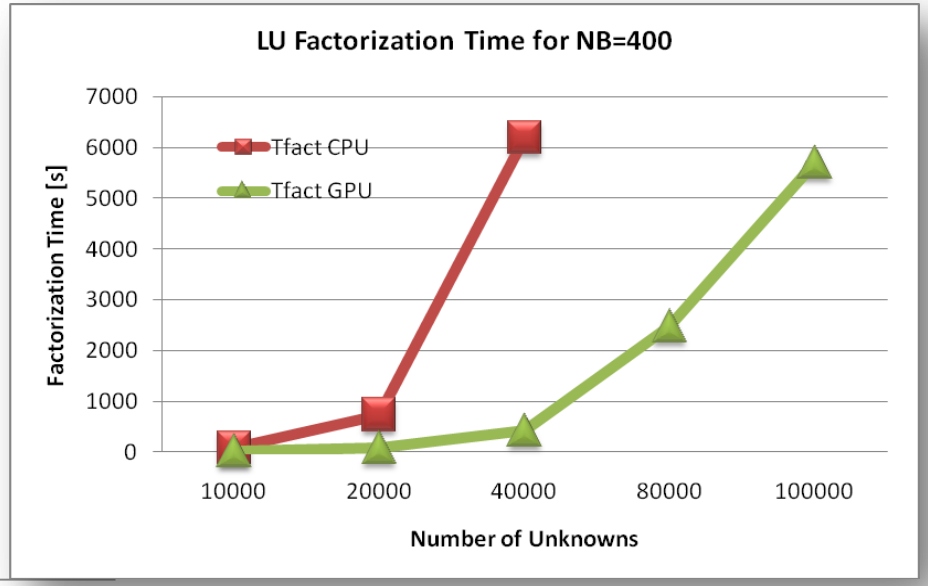
⇒ need fast parallel out-of-core solvers
⇒ Use GPUs as low-cost accelerators



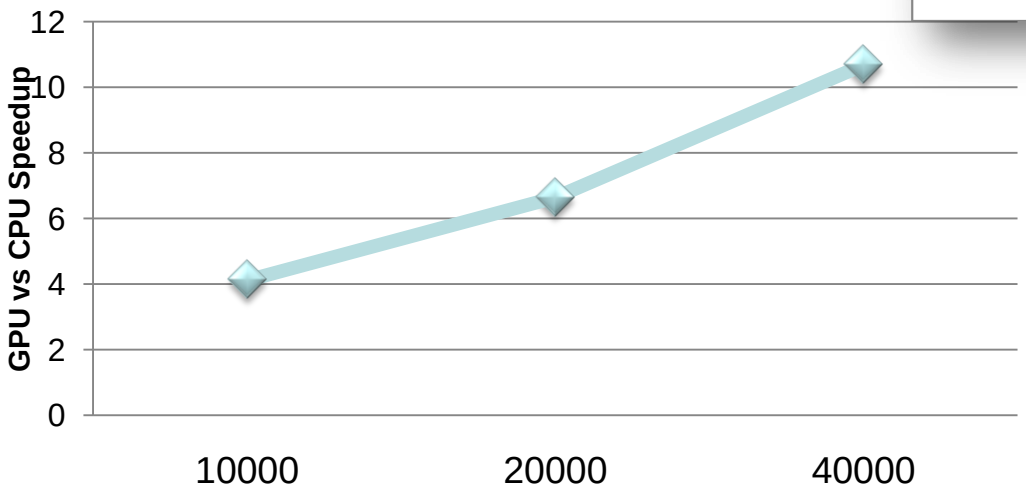


GPUs can significantly accelerate out-of-core LU decompositions

- Based on Azevedo/Dongarra OOC solver



GPU vs CPU Speedup for 8 way parallel OOC LU decomposition



Single precision, real

Summary/Conclusions

- GPUs offer large potential for accelerating scientific applications
- CUDA significantly simplifies code development
  - Still requires understanding of hardware
- GPULib enables GPU development from within VHLLs
  - Provides large set of vector operations with unified interface
  - Enables rapid development of GPU accelerated algorithms
  - No hardware knowledge required
- GPUs yields ~10x-40x speedup compared to CPU
- Fermi a big leap forward, both in performance and usability
  (to be released Q1/Q2 2010)
- Tech-X offers custom kernel development for maximum performance

Backup Slides

Related Projects at Tech-X

Some completed, some funded, some pending ...

GPULib Related Projects at Tech-X

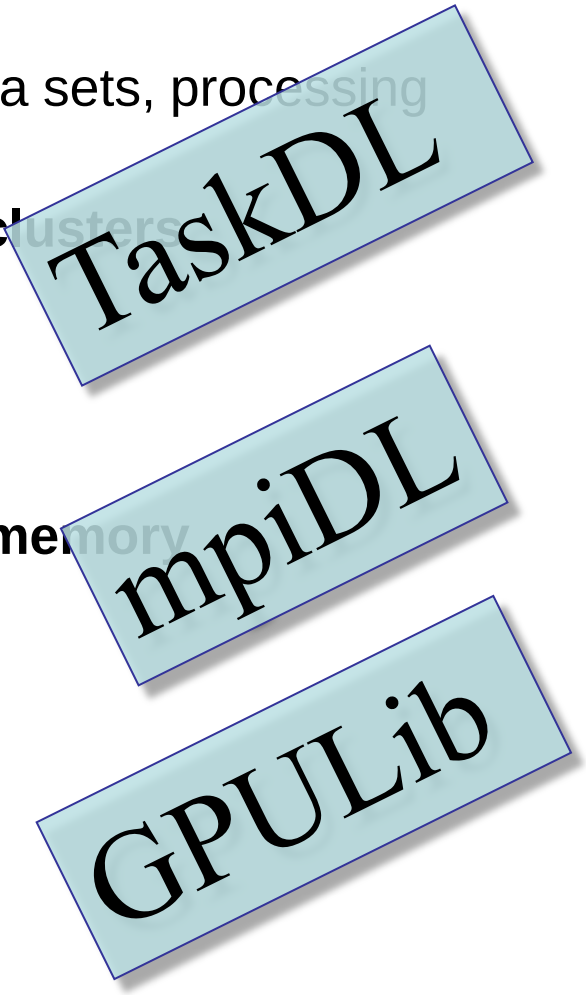
- FastDL – cluster computing in IDL (task farming and MPI)
- RDL – Remote Data Access via OpenDAP
- High-Performance/GPU computing in Space
- Parallel GPU accelerated dense linear solvers
- Radar/SAR processing on GPUs

Tech-X offers different tools for different HPC problems

- Processing large numbers of independent data sets, processing pipelines
  - **Task-Farming (incl. dependencies) on clusters**

- Process largest scale data sets
  - **Domain decomposition on distributed memory architectures**

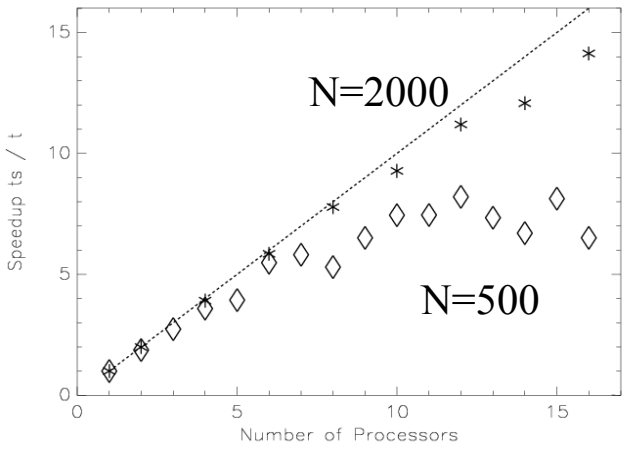
- Processing takes too long on single box
  - **Accelerator hardware (e.g. GPU)**



mpiDL: Code example

```
pro mpidl_demo
   myrank = MPIDL_Comm_rank()                    ; get processor id
   nprocs = MPIDL_Comm_size()                    ; get number of processors
   if myrank eq 0 then begin                     ; task for processor 0
      for i = 1, nprocs-1 do  begin
         message = MPIDL_Recv(count=1, source=i)          ; receive messages
         print, "Node   ", i, " sends ",  message         ; display the message
      end
   end else begin
     sendbuf = myrank                            ; build message to send
     MPIDL_Send, sendbuf, dest = 0               ; send this message to processor 0
   end
end
```

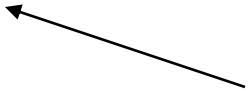
- Example: Full N^2 gravitational problem
  - Each processor computes force on subset of particles
  - Broadcasts forces to other processors



Writing optimized CUDA code can be painful

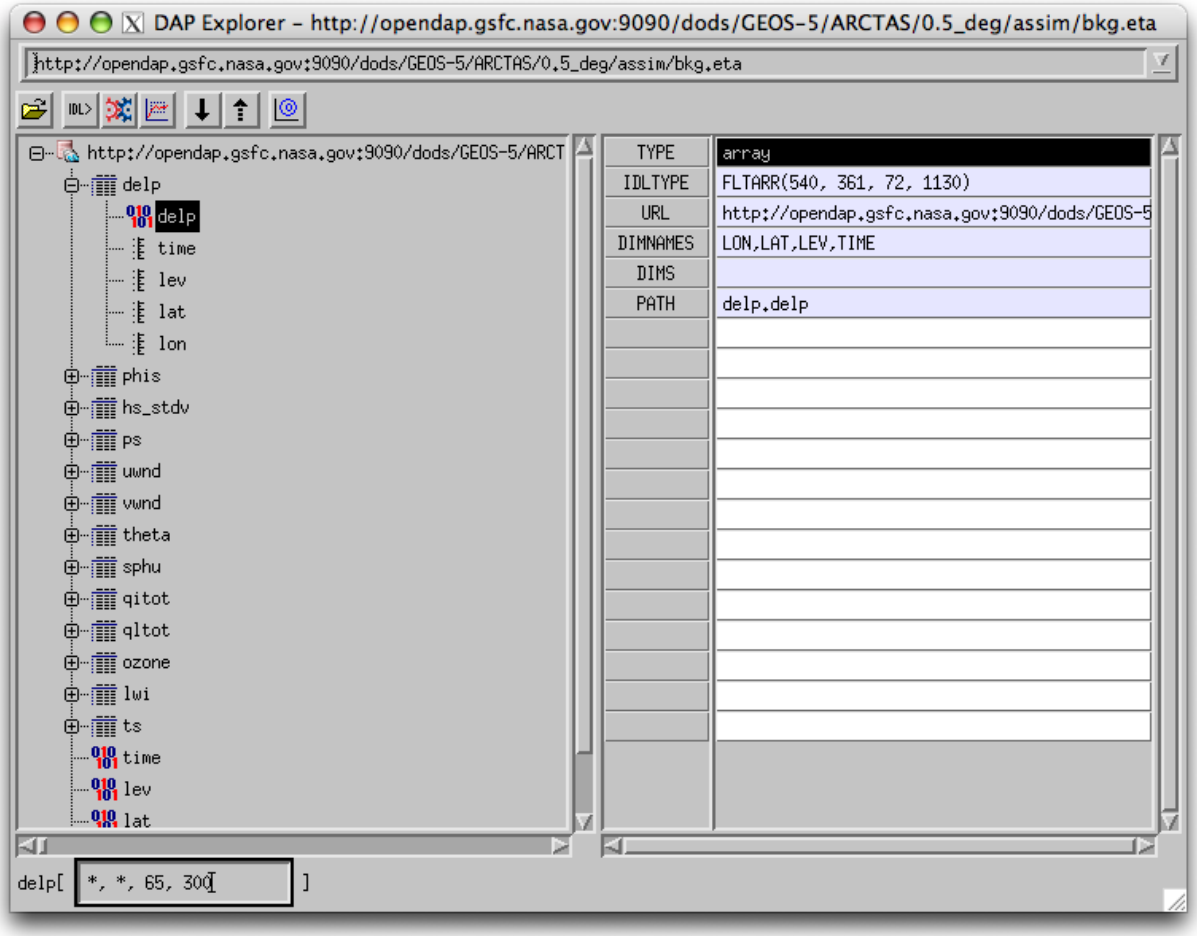
```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
	extern __shared__ int sdata[];
	unsigned int tid = threadIdx.x;
	unsigned int i = blockIdx.x*(blockSize*2) + tid;
	unsigned int gridSize = blockSize*2*gridDim.x;
	sdata[tid] = 0;
	while (i < n) { sdata[tid] += g_idata[i] + g_idata[i+blockSize]; i += gridSize; }
	__syncthreads();

	if (blockSize >= 512) { if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
	if (blockSize >= 256) { if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
	if (blockSize >= 128) { if (tid < 64) { sdata[tid] += sdata[tid + 64]; } __syncthreads(); }

	if (tid < 32) {
		if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
		if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
		if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
		if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
		if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
		if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
	}
	if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

Goal: Compute a vector of partial sums for a given vector



Manual loop unrolling

Remote Data Toolkit

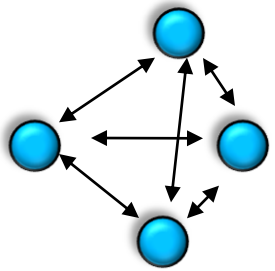


- Works with open standard DAP (Data Access Protocol)
- IDL Command line or GUI interface

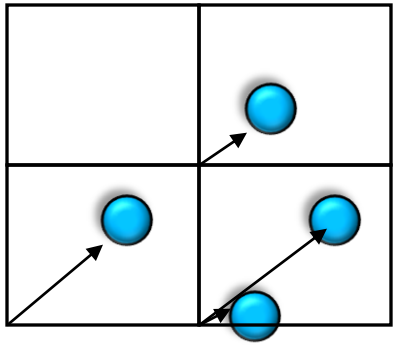
Funded by NASA SBIR Grant NNX08CA99P

Kinetic Plasma Models

- **Particle nature of plasma important**
  - E.g. non-thermal phase-space distributions
- **Particle-particle interaction too time consuming**
  - Compute Coulomb forces between particles
  - Computationally too demanding
    - $O(N^2)$ algorithm
  - Some projects may require it
    - Molecular dynamics

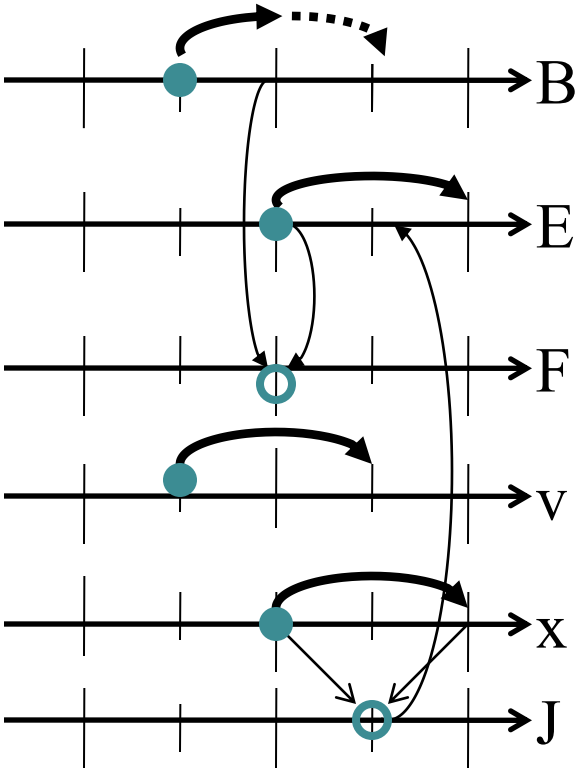


- **Particle-in-cell algorithm**
  - Particles: anywhere
  - Fields: discretized in space
  - Particle only interact with fields
    - Collective effects correct
  - O(N) algorithm -> Feasible!



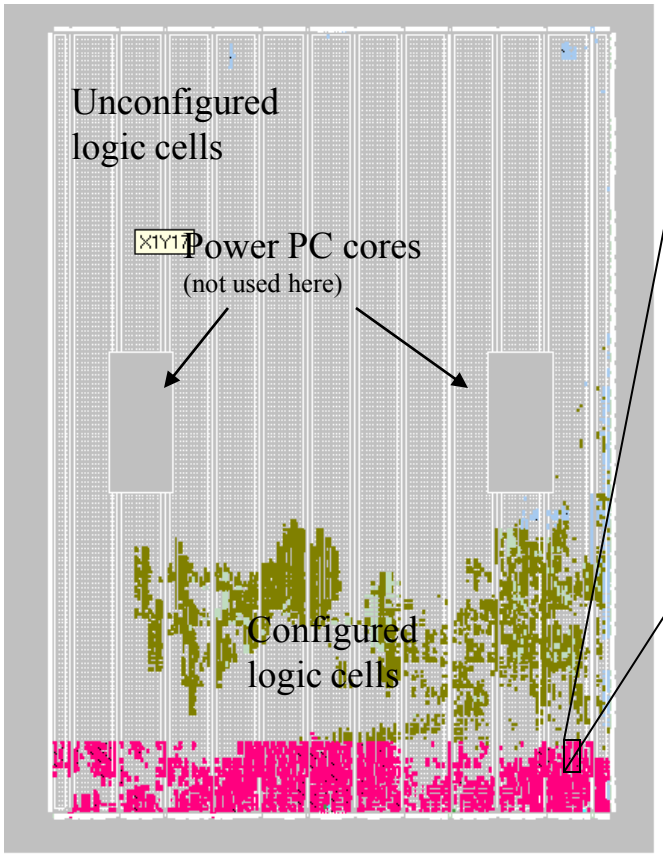
Particle-in-cell algorithm (PIC)

- **Field equations**
  - Initially known:
    - Solution to Poisson's equation, $\nabla E = \rho$
    - B-field divergence-free, $\nabla B = 0$
  - Dynamic Maxwell equations:
    - $dE/dt = \nabla \times B - J$
    - $dB/dt = -\nabla \times E$
- **Particle equations**
  - Newton-Lorentz:
    - $dv/dt = (q/m)(E - v \times B)$
- **Coupling fields-particles via current**
  - Determined by particle motion
  - Only dynamics equations need to be solved
  - Time-staggered leap frog
- **Self-consistent!**

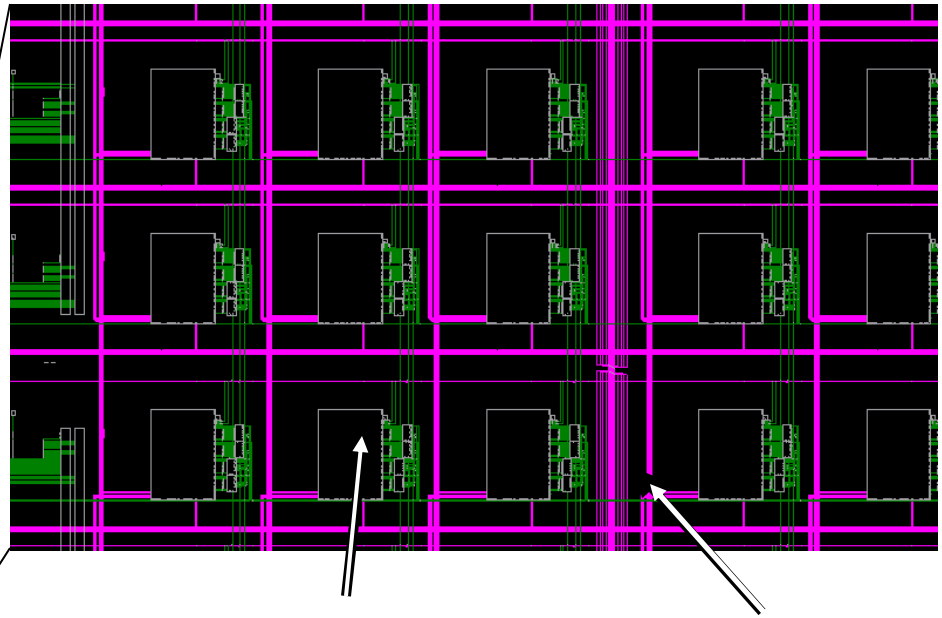


FPGA – Field Programmable Gate Arrays

Configurable matrix of logic and routes .. Yet another accelerator



Xilinx Virtex-II Pro FPGA



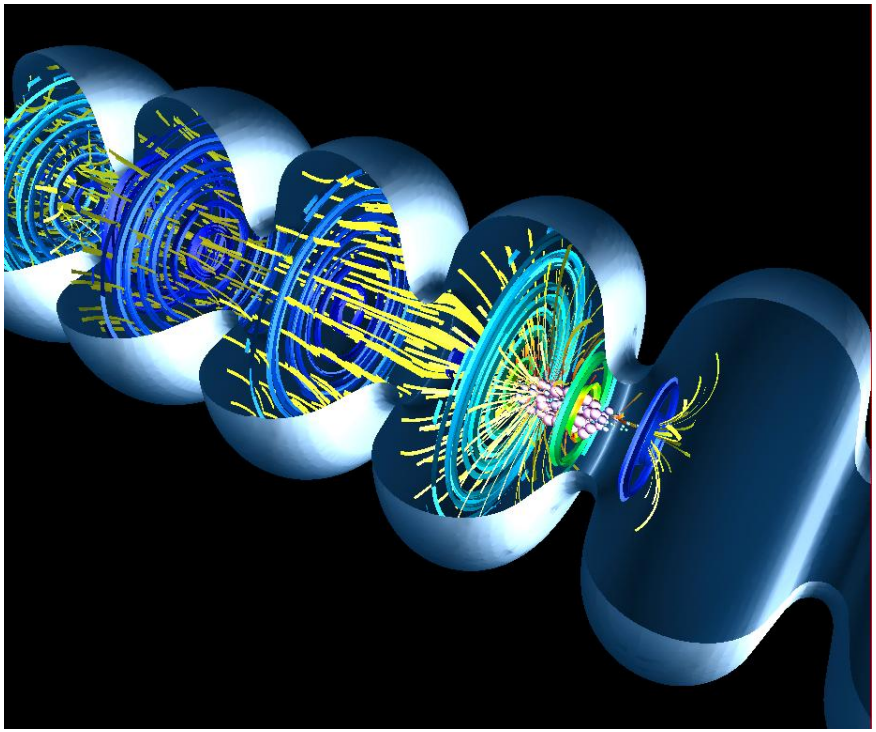
Logic cells
(Multipliers, Memory,
Look-up tables, shift registers,..)

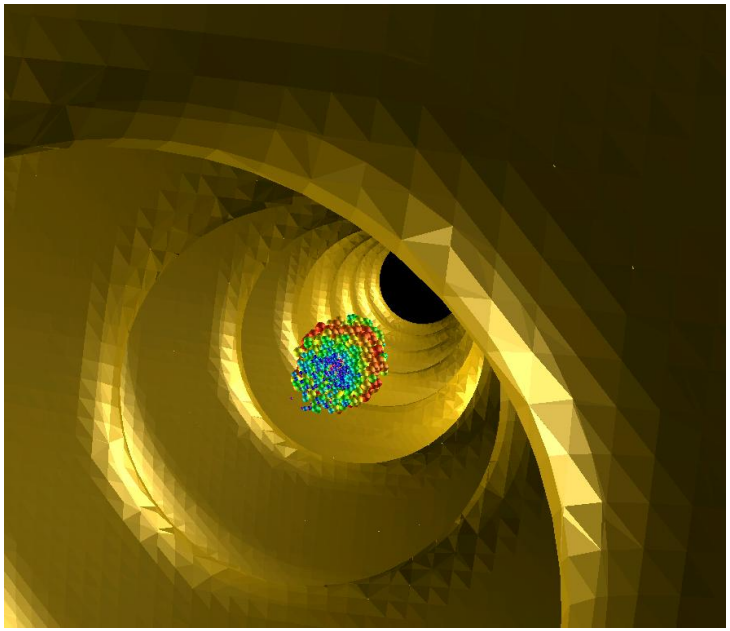
Routes

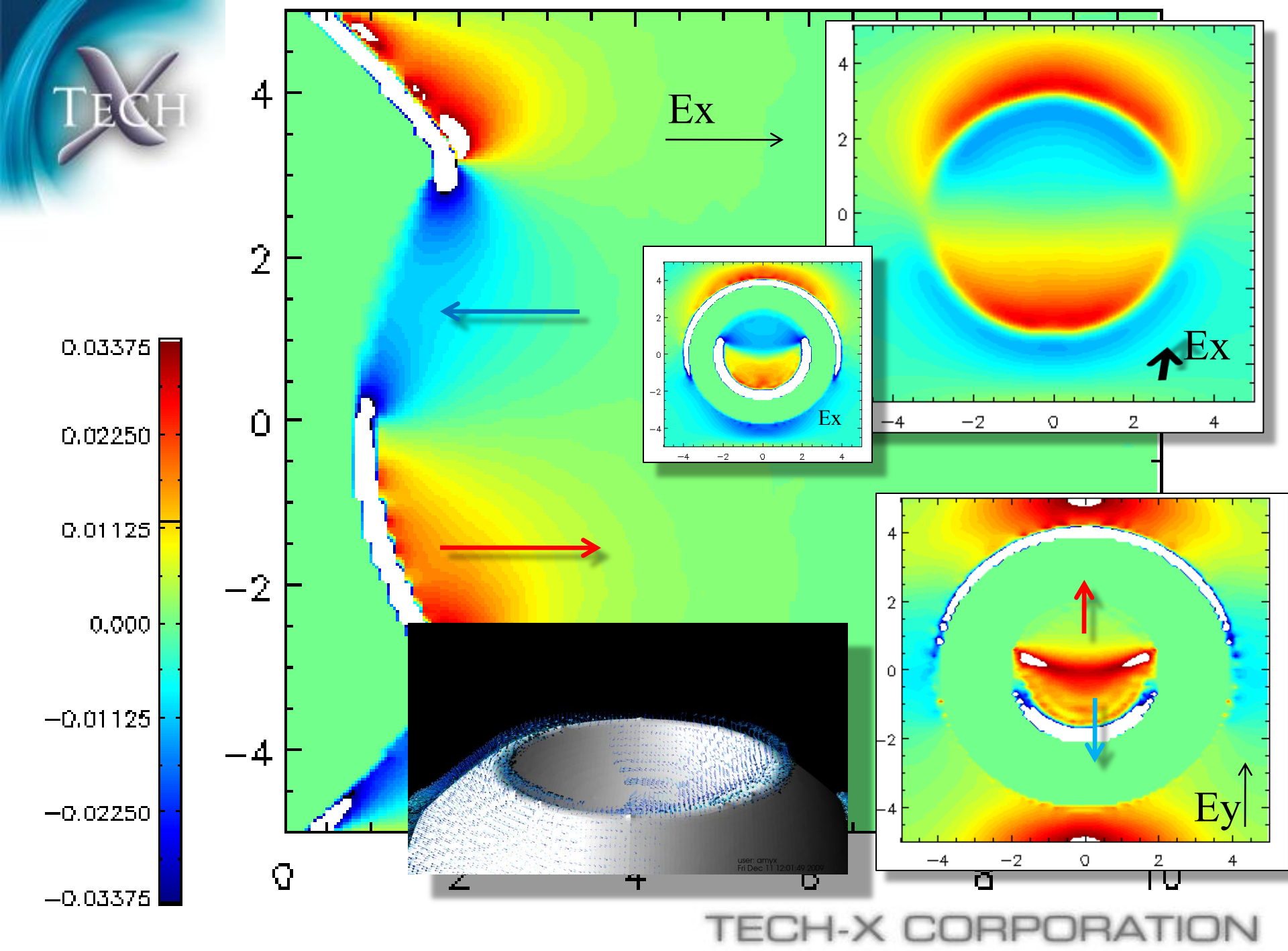
Logic/routes configured via
configuration file (bit-stream)

Main question: How to generate bit-streams??

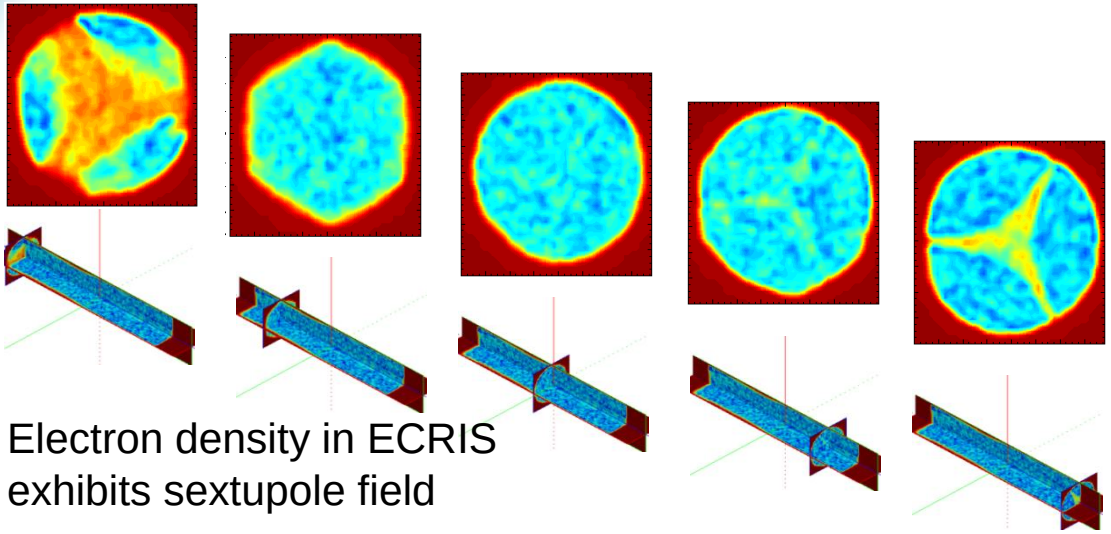
Examples of cut-cell FDTD simulations



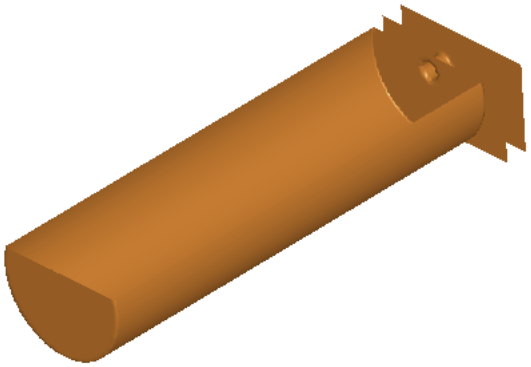



Ex
Ex
Ex
Ey
0.03375
0.02250
0.01125
0.000
−0.01125
−0.02250
−0.03375
TECH-X CORPORATION

VORPAL modeling of RF coupling into ECRIS plasma

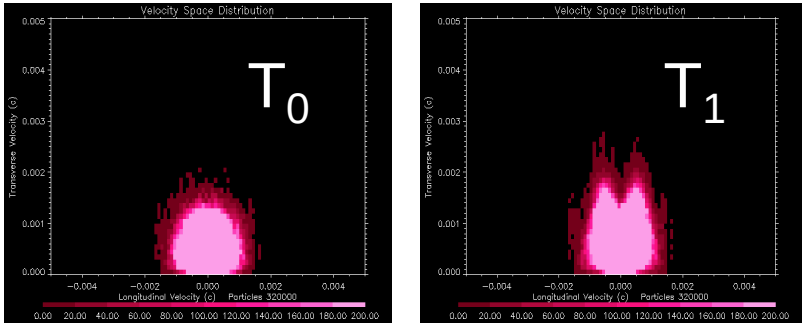


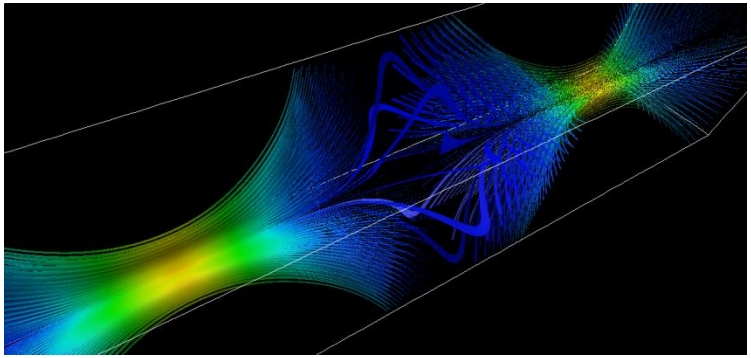
Electron density in ECRIS exhibits sextupole field



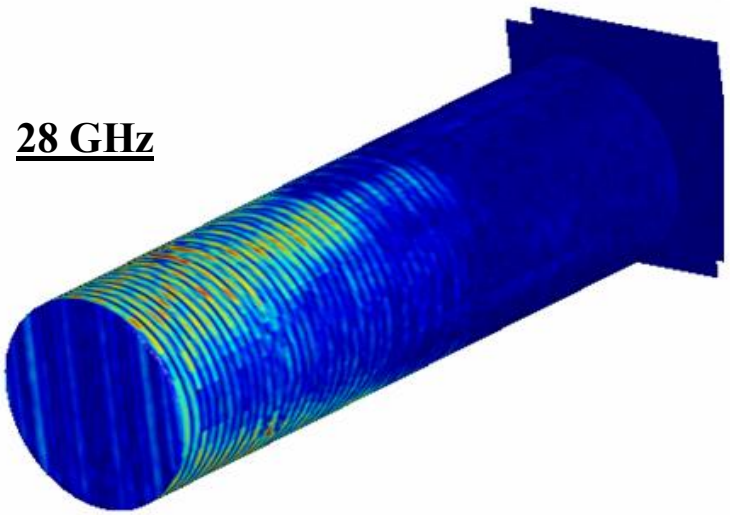
Device geometry with extraction optics and mag. bottlle configuration

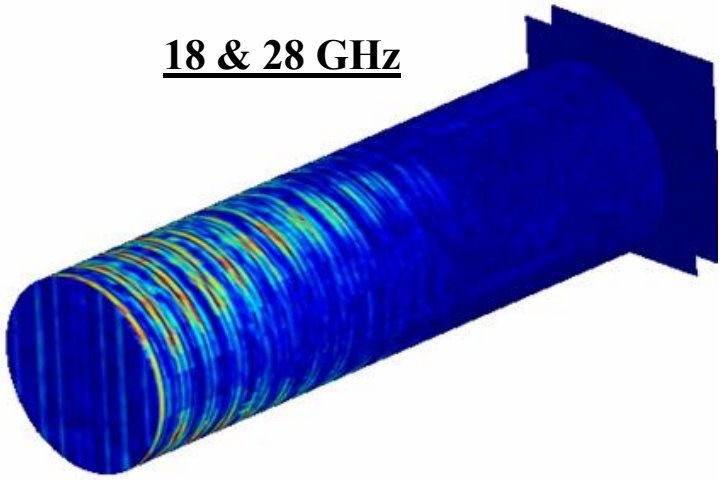
Resonant absorption of RF waves by electrons



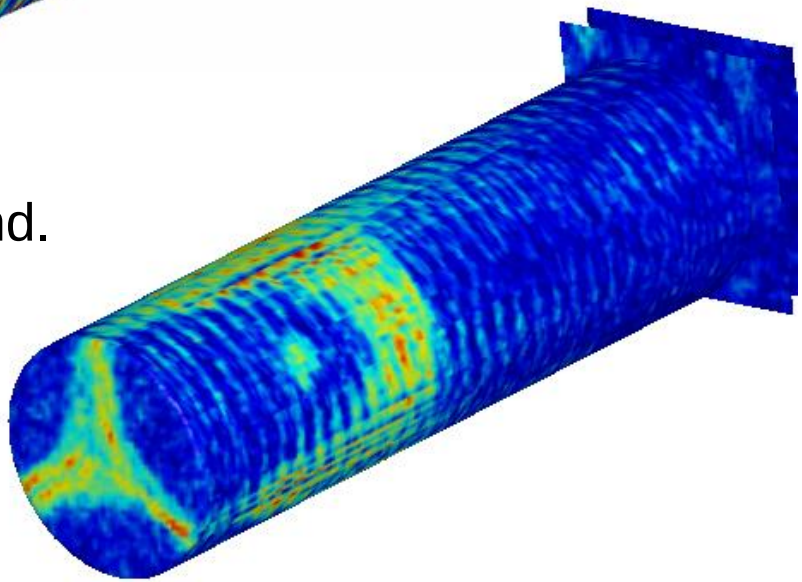


Single vs. dual frequency heating

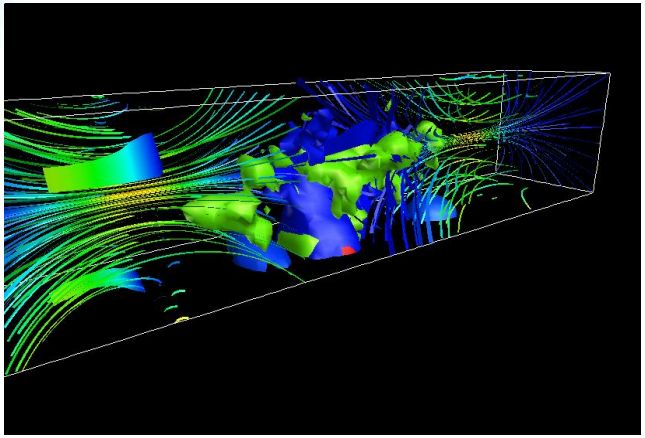


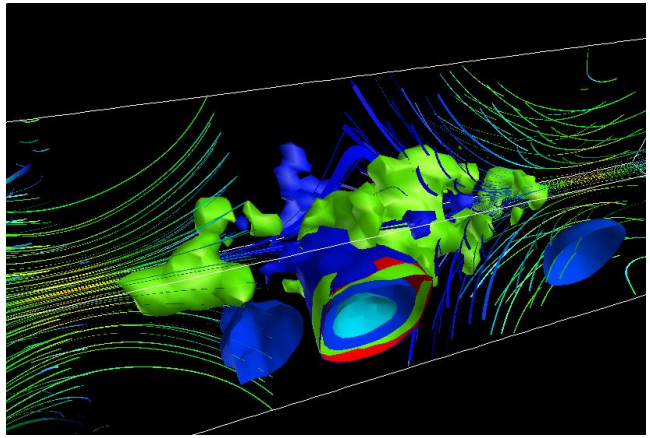


- Plane wave TE11 excitation from one end.
- Dual frequency shows 5 GHz beat
- Diagnostic of electron impact on surface

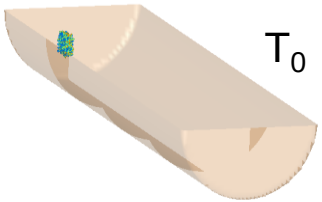


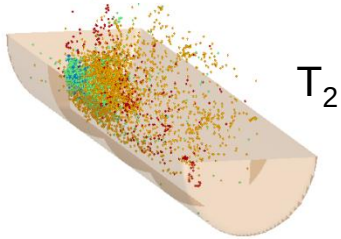
VORPAL's atomic physics model enable investigations of ionization cascade



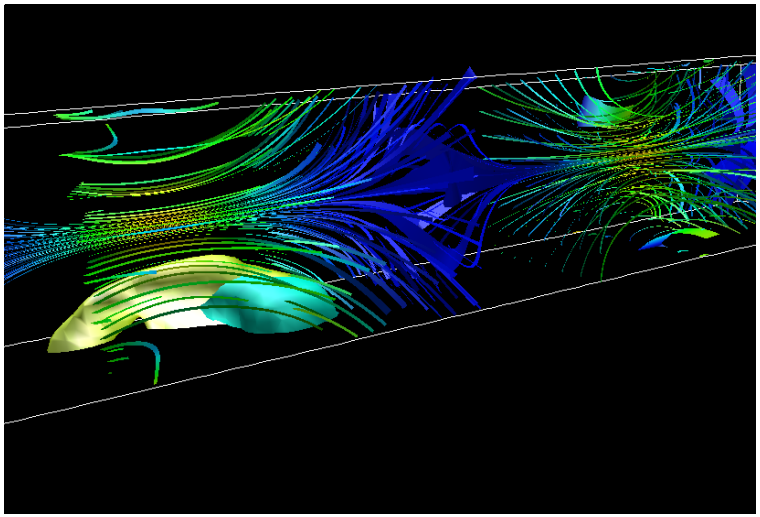


Loading in center of Mag bottle



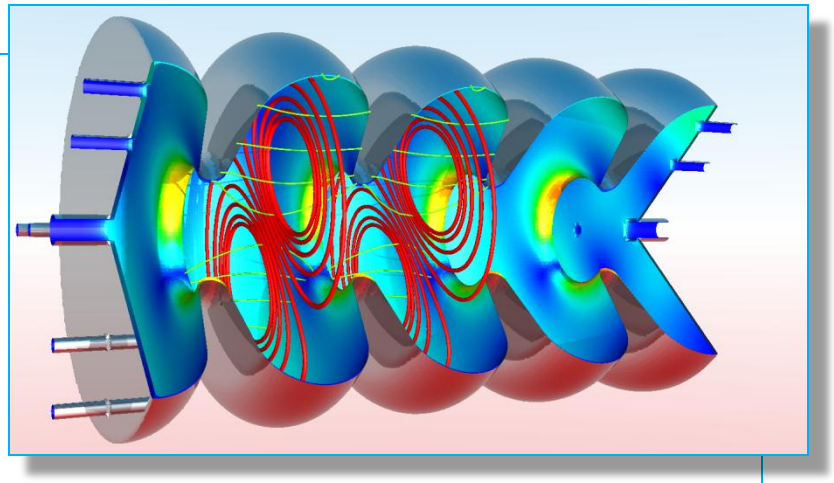


Distribution of
O, O^+, O^{2+}, O^{3+}



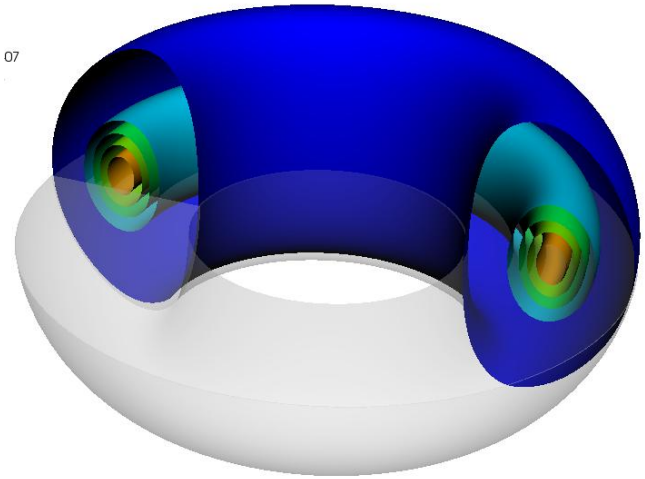
Loading close to bottle necks

Remote/Parallel Visualization



- Problem 1: Simulation output volume
- Problem 2: Simulation output location

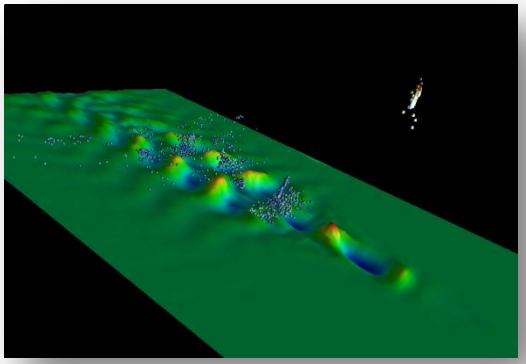
- Solution 1: Parallelism for analysis/redution
  - Use parallelism for data analysis/reduction not only for actual simulation
- Solution 2: Abandon remote X
  - No support for local accelerator hardware
  - Sometimes fairly chatty
- VisIT (Lawrence Livermore Lab) addresses both!
  - Parallel back-engine generates 3D primitives
  - Client-server architecture
  - Client can take advantage of 3D accelerator hardware
  - Output to high-end ray-tracers (eg Povray)

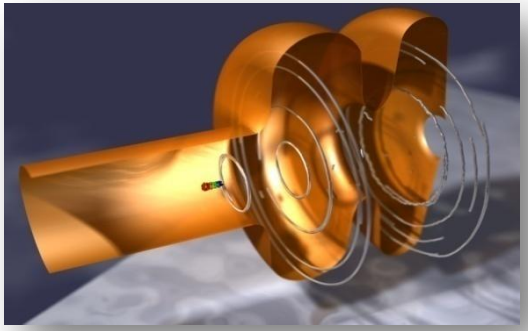


- Tech-X simulation output targets VisIT
  - Collaboration with VisIT developers to simplify basic viz.
  - Meta-information according to VizSchema

Tech-X Products

VORPAL: Plasma and EM modeling framework



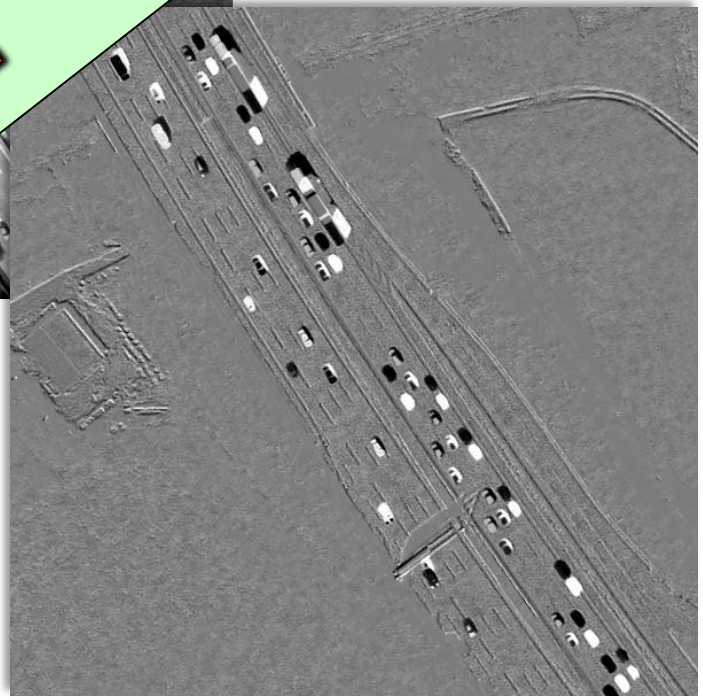


- Broad range of plasma and EM models
  - Particle-in-cell, Boltzmann fluid, Euler fluids
- Cut-cell algorithms for accurate electromagnetics in complex geometries
- Runs on largest-scale parallel supercomputers
  - Routinely on > 30,000 node concurrently
- Broad library of atomic physics, surface physics
  - Sputtering, secondary emission,
  - Ionization, recombination, nuclear reactions
- Widely used in government, industry and academia (all projects below with involvement of Tech-X personnel)
  - High-energy physics: Laser Wakefield Accelerators
  - Aerospace: Plasma engines, dielectric barrier discharges
  - Photonics: Bandgap structures, meta-materials
  - Surface interaction: Multi-pactoring, electron cloud effects
  - High-Power Microwave: Cavity optimization, Magnetrons, Gyrotrons
  - Semi-conductor manufacturing: Surface deposition reactors
  - Ion sources: Electron cyclotron resonance ion sources
  - High energy-density physics: laser fusion, inertial confinement fusion (ICF)

GPULib in ENVI: Principal Component Analysis

Speedup: 16x

Principal Component Analysis (PCA)



Data courtesy of
Dr. Mort Canty,
FZ Juelich, Germany

http://fwenvi-idl.blogspot.com/

Example: Database search

- Find closest match in 500k words with 128 characters each
- Less than 10ms
- CPU: ~200 ms

- GPULib 1: 500k dot-products
  - Need test vector on GPU
  - Vectors short
  - Huge number of kernel invocations

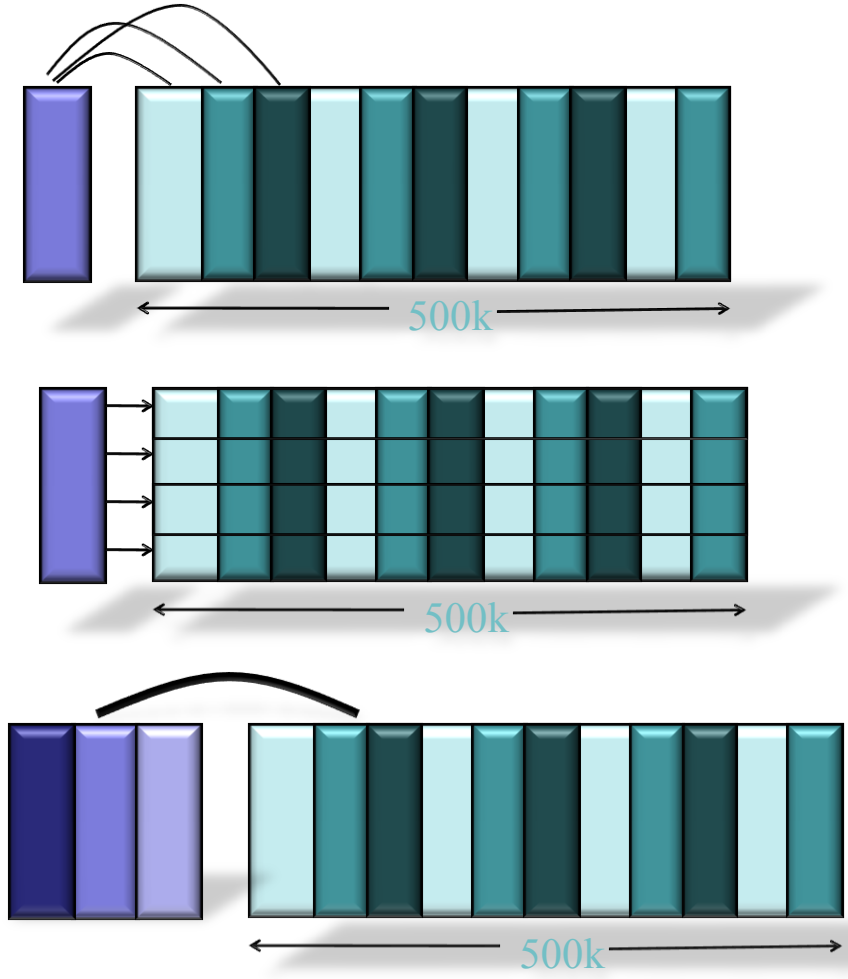
  => Bad idea

- GPULib 2: 128 accumulations
  - No need to transfer entire vector
  - Large vectors
  - Smaller number of kernel invocations

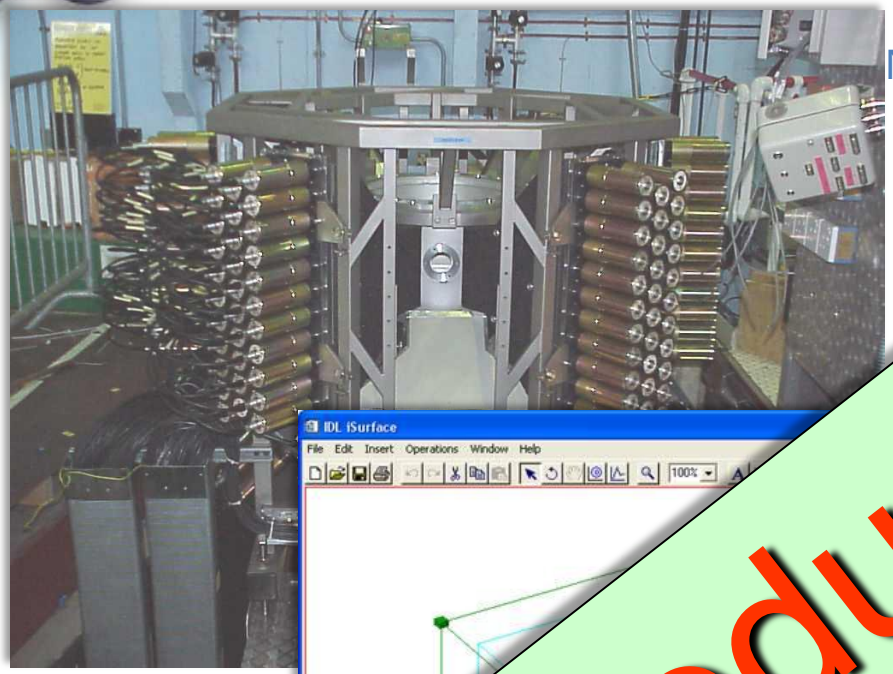
  => ~27 ms

- Hand crafted implementation
  - Transfer data to GPU
  - Perform 128 dot products concurrently

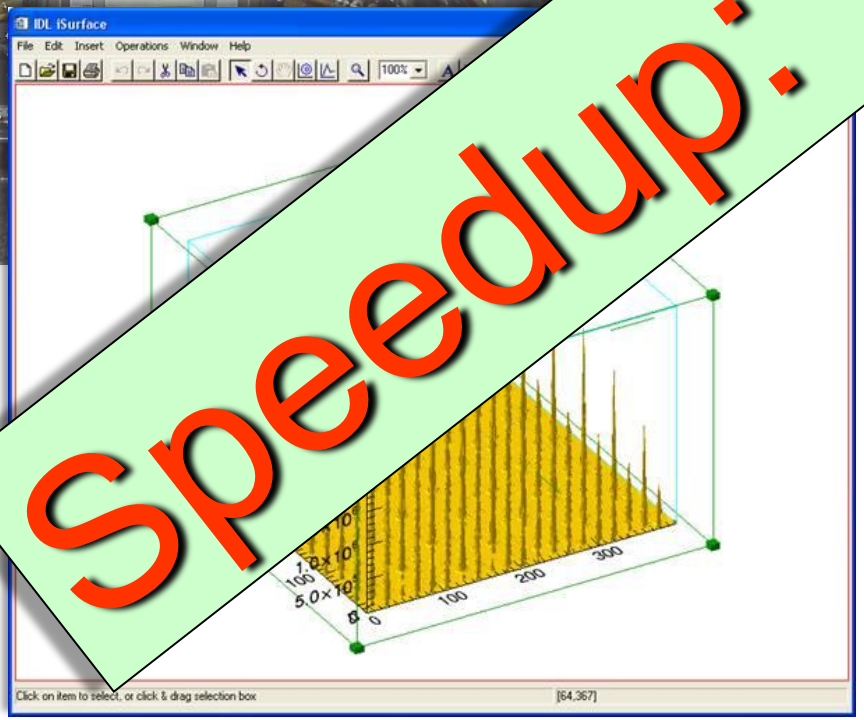
  => < 8 ms (Tesla 8-series GPU)



GPULib example: Simulation



Neutron scattering e eriment



Speedup: 10x

se simulation written in IDL to compute location of scattering maxima (Bragg peaks)

Data courtesy of
Dr. Matthias Gutmann,
Rutherford Appleton
Research Lab, UK