An Introduction to OpenACC: There is no such thing as a free lunch, but some things taste better than others.

OLCF
Oak Ridge Leadership Computing Facility

OpenACC®
DIRECTIVES FOR ACCELERATORS

Bronson Messer

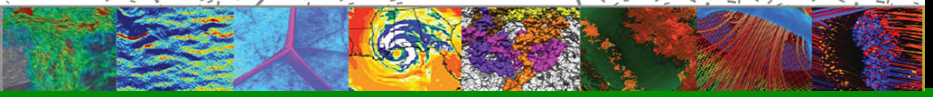
Scientific Computing Group

National Center for Computational Sciences

Theoretical Astrophysics Group
Oak Ridge National Laboratory

Department of Physics & Astronomy
University of Tennessee

Extreme Scale Computing
TRAINING PROGRAM



Credits

- Many of the slides here come directly from several people

- Mat Colgrove (PGI)
- Jeff Larkin (NVIDIA)
- John Levesque (Cray)

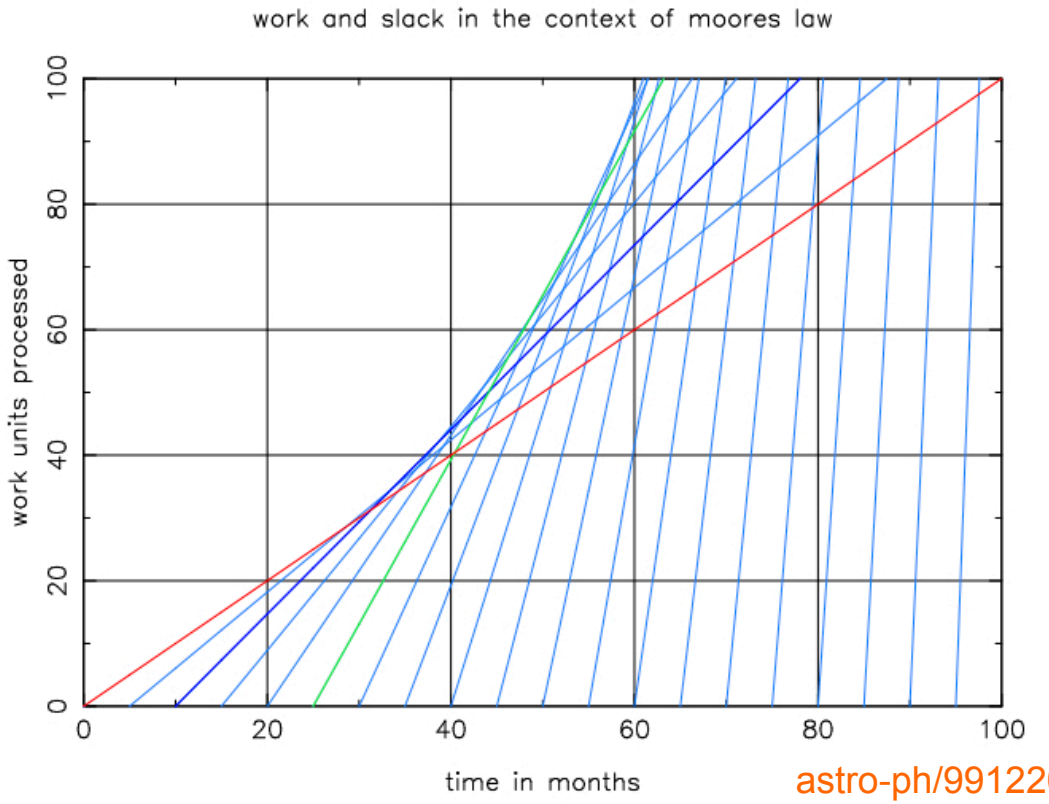
The Effects of Moore's Law and Slacking [1] on Large Computations

Chris Gottbrath, Jeremy Bailin, Casey Meakin, Todd Thompson, J.J. Charfman
Steward Observatory, University of Arizona

[1]This paper took 2 days to write

Abstract

We show that, in the context of Moore's Law, overall productivity can be increased for large enough computations by 'slacking' or waiting for some period of time before purchasing a computer and beginning the calculation.



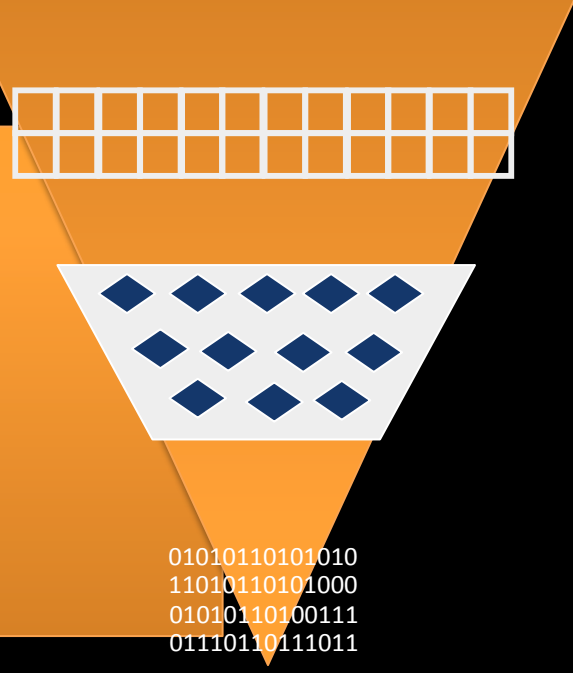
astro-ph/9912202

The future is NOW.

- All near-future systems will have a secondary memory that is as large as we currently require; however, it will not have high bandwidth to the principal computational engine.
- There will be a smaller, faster memory that will supply the principal compute engine.
- While system software may manage the two memories for the user, the user will have to manage these disparate memories to achieve maximum performance

Hierarchical Parallelism

- MPI parallelism between nodes (or PGAS)
- On-node, SMP-like parallelism via threads (or subcommunicators, or…)
- Vector parallelism
  - SSE/AVX/etc on CPUs
  - GPU threaded parallelism



- **Exposure of unrealized parallelism is essential to exploit <u>all</u> near-future architectures.**
- **Uncovering unrealized parallelism and improving data locality improves the performance of even CPU-only code.**
  - **E.g. in CAAR we found that the median (mode) speedup on CPU-only was ~2x**

So how should we program for these new systems?

- What to do – Good Threading (OpenMP)
  - Must do high level threading
  - Thread must access close shared memory rather than distant shared memory
  - Load Balancing
- What to do – Good Vectorization
  - Vectorization advantage allows for introducing overhead to vectorize
    - Vectorization of Ifs
      - Conditional vector merge (too many paths??)
      - Gather/scatter (Too much data motion??)
      - Identification of strings

http://openacc.org/

- A common directive programming model for today's GPUs
  - Announced at SC11 conference
  - Offers portability between compilers
    - Drawn up by: NVIDIA, Cray, PGI, CAPS
    - Multiple compilers offer portability, debugging, permanence
  - Works for Fortran, C, C++
    - Standard available at www.OpenACC-standard.org
    - Initially implementations targeted at NVIDIA GPUs
- Current version: 1.0 (November 2011)
- Compiler support:
  - Cray CCE: complete 1.0
  - PGI Accelerator: complete 1.0 + extensions
  - CAPS: complete 1.0 + extensions


The OpenACC™ API
QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

CAPS
CRAY THE SUPERCOMPUTER COMPANY
NVIDIA
PGI

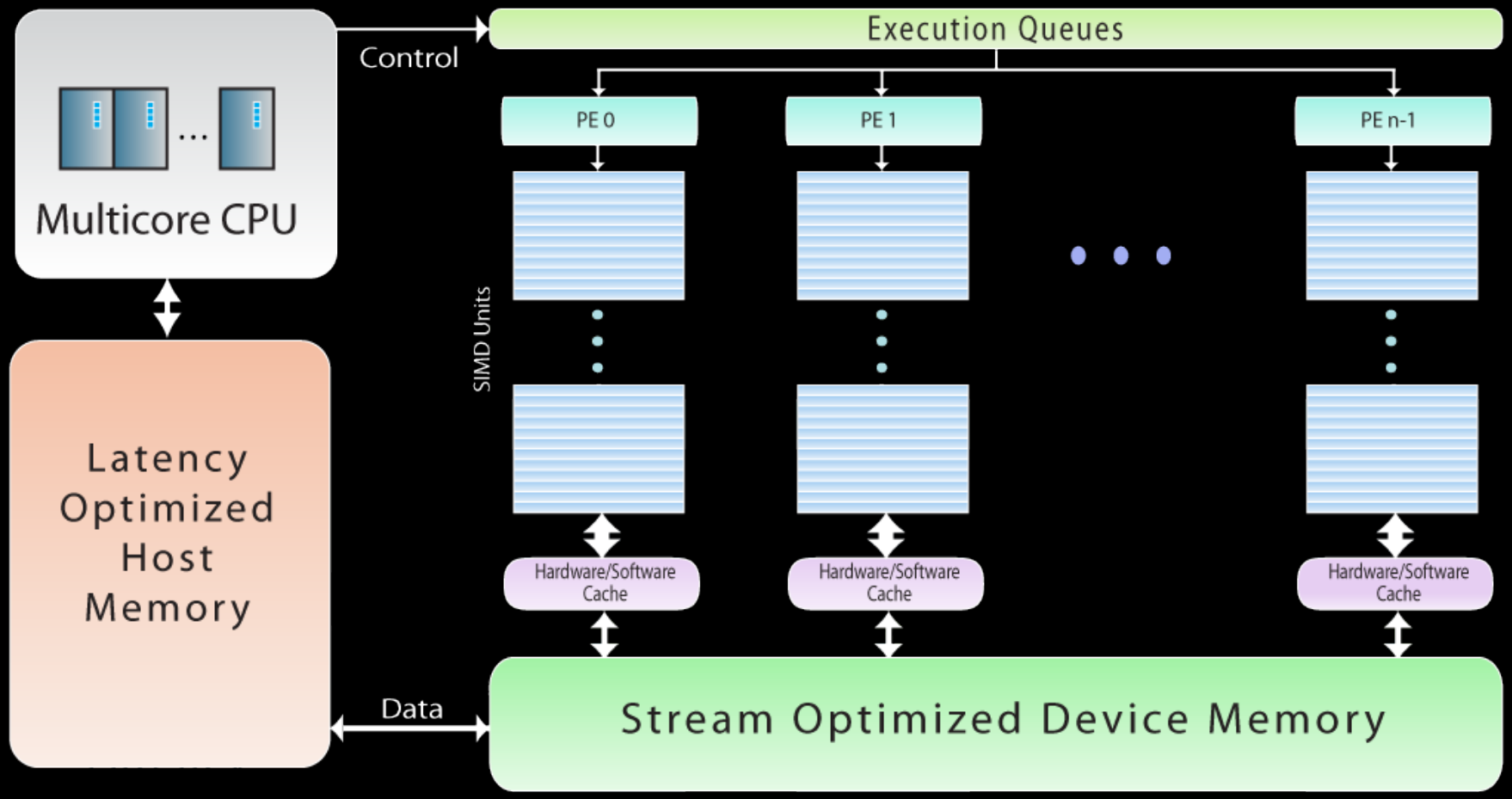
Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.

Risk factors

- Will there be machines to run my OpenACC code on?
  - Now? Lots of Nvidia GPU accelerated systems
    - Cray XK7s: CSCS tödi, HLRS hermit, ORNL titan…
    - Lots of other GPU machines in Top100 (OpenACC is multi-vendor)
  - Future? OpenACC can be targeted at other accelerators
    - PGI and CAPS already target Intel Xeon Phi, AMD GPUs
  - Plus you can always run on CPUs using same codebase
- Will OpenACC continue?
  - Support? Cray and PGI (at least) are committed to support OpenACC
    - Lots of big customer pressure to continue to run OpenACC
  - Develop? OpenACC committee now finalising v2.0 of standard (more on this later)
    - Lots of new partners joined committee at end of last year
- Will OpenACC be superseded by something else?
  - Auto-accelerating compilers? If only!
    - Never really managed it for threading real HPC applications on the CPU
    - Data locality adds to the challenge
  - OpenMP accelerator directives? OpenACC work not wasted
    - Very similar programming model; can transition easily
    - Cray (co-chair), PGI very active in OpenMP accelerator subcommittee

OpenACC Abstract Machine Architecture



PGI Accelerator Directive-based Compilers

```
#pragma acc kernels loop
  for( i = 0; i < nrows; ++i ){
    float val = 0.0f;
    for( d = 0; d < nzeros; ++d ){
      j = i + offset[d];
      if( j >= 0 && j < nrows )
        val += m[i+nrows*d] * v[j];
    }
    x[i] = val;
  }
```

compile

```
matvec:
        subq  $328, %rsp
...
        call  __pgi_cu_alloc
...
        call  __pgi_cu_uploadx
...
        call  __pgi_cu_launch2
...
        call  __pgi_cu_downloadx
...
        call  __pgi_cu_free
...
```

+

```
    .entry  matvec_14_gpu( ...
    .reg  .u32 %r<70> ...
    cvt.s32.u32  %r1, %tid.x;
    mov.s32      %r2, 0;
    setp.ne.s32  $p1, %r1, %r2
    cvt.s32.u32  %r3, %ctaid.x;
    cvt.s32.u32  %r4, %ntid.x;
    mul.lo.s32   %r5, %r3, %r4;
    @%p1 bra     $Lt_0_258;
    st.shared.s32 [__i2s], %r5
$Lt_0_258:
    bar.sync     0;
...
```

Link

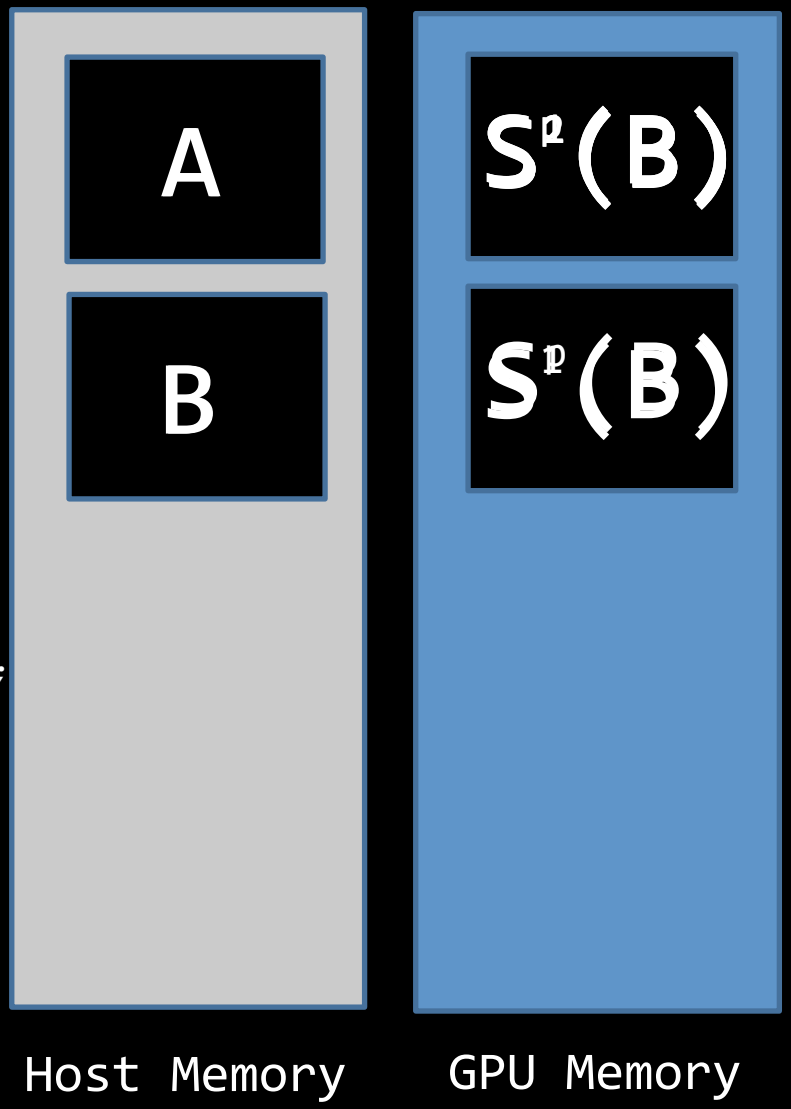
Unified Object

execute

… no change to existing makefiles, scripts, IDEs, programming environment, etc.

PGI Accelerator OpenACC example

```
#pragma acc data \
        copy(b[0:n][0:m]) \
        create(a[0:n][0:m])
{
for (iter = 1; iter <= p; ++iter){
  #pragma acc kernels
  {
  for (i = 1; i < n-1; ++i){
    for (j = 1; j < m-1; ++j){
      a[i][j]=w0*b[i][j]+
              w1*(b[i-1][j]+b[i+1][j]+
                  b[i][j-1]+b[i][j+1])+
              w2*(b[i-1][j-1]+b[i-1][j+1]+
                  b[i+1][j-1]+b[i+1][j+1]);
  } }
  for( i = 1; i < n-1; ++i )
    for( j = 1; j < m-1; ++j )
      b[i][j] = a[i][j];
  }
}
}
```



Why use OpenACC Directives?

- Productivity
  - Higher level programming model
  - *a la* OpenMP
- Portability
  - ignore directives, portable to the host
  - portable across different accelerators
  - *performance portability*
- Performance feedback

Matrix Multiply Source Code Size Comparison:

```
void
__global__ void matrixMul( float* C, float* A, float* B, int wA, int wB)
{
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd   = aBegin + wA - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * wB;
    float Csub = 0;
    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep) {
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];
        AS(ty, tx) = A[a + wA * ty + tx];
        BS(ty, tx) = B[b + wB * ty + tx];
        __syncthreads();
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += AS(ty, k) * BS(k, tx);
        __syncthreads();
    }
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}

void
domatmul( float* C, float* A, float* B, unsigned int hA, unsigned int wA , unsigned wB )
{
    unsigned int size_A = WA * HA;
    unsigned int mem_size_A = sizeof(float) * size_A;
    unsigned int size_B = WB * HB;
    unsigned int mem_size_B = sizeof(float) * size_B;
    unsigned int size_C = WC * HC;
    unsigned int mem_size_C = sizeof(float) * size_C;
    float *d_A, *d_B, *d_C;

    cudaMalloc((void**) &d_A, mem_size_A);
    cudaMalloc((void**) &d_B, mem_size_B);
    cudaMalloc((void**) &d_C, mem_size_C);
    cudaMemcpy(d_A, h_A, mem_size_A, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, mem_size_B, cudaMemcpyHostToDevice);

    dim3 threads(BLOCK_SIZE, BLOCK_SIZE);

    dim3 grid(WC / threads.x, HC / threads.y);

    matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA, WB);

    cudaMemcpy(h_C, d_C, mem_size_C, cudaMemcpyDeviceToHost);

    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```

```
void
computeMM0_saxpy(float C[][WB], float A[][WA], float B[][WB],
                 int hA, int wA, int wB)
{
#pragma acc region
{
#pragma acc for parallel vector(16) unroll(4)
    for (int i = 0; i < hA; ++i) {
        for (int j = 0; j < wB; ++j) {
            C[i][j] = 0.0 ;
        }
        for (int k = 0; k < wA; ++k) {
            for (int j = 0; j < wB; ++j) {
                C[i][j] = C[i][j]+A[i][k]*B[k][j];
            }
        }
    }
}
}
```

```
void matrixMulGPU(cl_uint ciDeviceCount, cl_mem h_A, float* h_B_data,
                  unsigned int mem_size_B, float* h_C )
{
    cl_mem d_A[MAX_GPU_COUNT];
    cl_mem d_C[MAX_GPU_COUNT];
    cl_mem d_B[MAX_GPU_COUNT];

    cl_event GPUDone[MAX_GPU_COUNT];
    cl_event GPUExecution[MAX_GPU_COUNT];

    // Create buffers for each GPU
    // Each GPU will compute sizePerGPU rows of the result
    int sizePerGPU = HA / ciDeviceCount;

    int workOffset[MAX_GPU_COUNT];
    int workSize[MAX_GPU_COUNT];

    workOffset[0] = 0;
    for(unsigned int i=0; i < ciDeviceCount; ++i)
    {
        // Input buffer
        workSize[i] = (i != (ciDeviceCount - 1)) ? sizePerGPU : (HA - workOffset[i]);

        d_A[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY, workSize[i] * sizeof(float) * WA, NULL,NULL);

        // Copy only assigned rows from host to device
        clEnqueueCopyBuffer(commandQueue[i], h_A, d_A[i], workOffset[i] * sizeof(float) * WA,
                            0, workSize[i] * sizeof(float) * WA, 0, NULL, NULL);

        // create OpenCL buffer on device that will be initiatlize from the host memory on first use
        // on device
        d_B[i] = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                                mem_size_B, h_B_data, NULL);

        // Output buffer
        d_C[i] = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,  workSize[i] * WC * sizeof(float), NULL,NULL);

        // set the args values
        clSetKernelArg(multiplicationKernel[i], 0, sizeof(cl_mem), (void *) &d_C[i]);
        clSetKernelArg(multiplicationKernel[i], 1, sizeof(cl_mem), (void *) &d_A[i]);
        clSetKernelArg(multiplicationKernel[i], 2, sizeof(cl_mem), (void *) &d_B[i]);
        clSetKernelArg(multiplicationKernel[i], 3, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );
        clSetKernelArg(multiplicationKernel[i], 4, sizeof(float) * BLOCK_SIZE *BLOCK_SIZE, 0 );

        if(i+1 < ciDeviceCount)
            workOffset[i + 1] = workOffset[i] + workSize[i];
    }
    // Execute Multiplication on all GPUs in parallel
    size_t localWorkSize[] = {BLOCK_SIZE, BLOCK_SIZE};
    size_t globalWorkSize[] = {shrRoundUp(BLOCK_SIZE, WC), shrRoundUp(BLOCK_SIZE, workSize[0])};
    // Launch kernels on devices
        for(unsigned int i = 0; i < ciDeviceCount; i++)
        {
                        // Multiplication - non-blocking execution
                        globalWorkSize[1] = shrRoundUp(BLOCK_SIZE, workSize[i]);
                        clEnqueueNDRangeKernel(commandQueue[i], multiplicationKernel[i], 2, 0, globalWorkSize, localWorkSize,
                                               0, NULL, &GPUExecution[i]);
                    }
        for(unsigned int i = 0; i < ciDeviceCount; i++)
    {
                clFinish(commandQueue[i]);
        }
    for(unsigned int i = 0; i < ciDeviceCount; i++)
    {
// Non-blocking copy of result from device to host
        clEnqueueReadBuffer(commandQueue[i], d_C[i], CL_FALSE, 0, WC * sizeof(float) * workSize[i],
                            h_C + workOffset[i] * WC, 0, NULL, &GPUDone[i]);
    }
// CPU sync with GPU
    clWaitForEvents(ciDeviceCount, GPUDone);

// Release mem and event objects
    for(unsigned int i = 0; i < ciDeviceCount; i++)
    {
        clReleaseMemObject(d_A[i]);
        clReleaseMemObject(d_C[i]);
        clReleaseMemObject(d_B[i]);
        clReleaseEvent(GPUExecution[i]);
        clReleaseEvent(GPUDone[i]);
    }
}
__kernel void
matrixMul( __global float* C, __global float* A, __global float* B,
           __local float* As, __local float* Bs)
{
    int bx = get_group_id(0), tx = get_local_id(0);
    int by = get_group_id(1), ty = get_local_id(1);
    int aEnd   = WA * BLOCK_SIZE * by + WA - 1;

    float Csub = 0.0f;

    for (int a = WA*BLOCK_SIZE*by , b = BLOCK_SIZE * bx;
         a <= aEnd; a += BLOCK_SIZE, b += BLOCK_SIZE*WB) {
        As[tx + ty * BLOCK_SIZE] = A[a + WA * ty + tx];
        Bs[tx + ty * BLOCK_SIZE] = B[b + WB * ty + tx];
        barrier(CLK_LOCAL_MEM_FENCE);
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[k + ty * BLOCK_SIZE]*Bs[tx + k * BLOCK_SIZE] ;
        barrier(CLK_LOCAL_MEM_FENCE);
    }
    C[get_global_id(1) * get_global_size(0) + get_global_id(0)] = Csub;

}
```

Directives

CUDA C

OpenCL

Kernels Construct

- C

```
#pragma acc kernels clause...
{
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
```

- Fortran

```
!$acc kernels clause...
     do i = 1,n
          r(i) = a(i) * 2.0
     enddo
!$acc end kernels
```

Parallel Construct

- C

```
#pragma acc parallel clause...
{
    #pragma acc loop gang vector
    for( i = 0; i < n; ++i ) r[i] = a[i]*2.0f;
}
```

- Fortran

```
!$acc parallel clause...
!$acc loop gang vector
    do i = 1,n
       r(i) = a(i) * 2.0
    enddo
!$acc end parallel
```

Kernels vs. Parallel

- Kernels Construct
  - Derived from PGI Accelerator Model
  - More implicit giving the compiler more freedom to create optimal code for a given accelerator
  - Works best for tightly nested loops
  - May require some additional 'hints' to the compiler
    - i.e. C99 restrict keyword
- Parallel Construct
  - Based on OpenMP "workshare"
  - Create parallel gangs that execute redundantly
  - Each gang executes a portion of a work-sharing loop
  - More explicit requiring some user intervention

http://www.pgroup.com/lit/articles/insider/v4n2a1.htm

Loop Directive

- C

```
#pragma acc loop clause...
for( i = 0; i < n; ++i ){
    ....
}
```

- Fortran

```
!$acc loop clause...
do i = 1, n
```

Note: Compute Constructs and the Loop directive may be combined

Loop Directive Clauses

- `independent`
  - use with care, overrides compiler analysis for dependence, private variables (kernels only, implied with parallel)
- `private(` *`list`* `)`
  - private data for each iteration of the loop
- `reduction(` *`red:var`* `)`
  - reduction across the loop

- Scheduling Clauses
  - `vector` or `vector(`*`width`*`)`
  - `gang` or `gang(`*`width`*`)`
  - `worker` or `worker(`*`width`*`)`
  - `seq`

Important Terminology

- Gang
  - The highest level of parallelism, equivalent to CUDA Threadblock. (num_gangs => number of threadblocks)
  - A “gang” loop affects the “CUDA Grid”
- Worker
  - A member of the gang, equivalent to CUDA thread within a threadblock (num_workers => threadblock size)
  - A “worker” loop affects the “CUDA Threadblock”
- Vector
  - Tightest level of SIMT/SIMD/Vector parallelism, equivalent to CUDA warp or SIMD vector length (vector_length should be a multiple of warp size)
  - A ‘vector” loop affects the SIMT parallelism

- Declaring these on particular loops in your loop nest will affect the decomposition of the problem to the hardware

Stupid Loop-Scheduling Tricks

- `!$acc loop gang`
  - runs in 'gang' mode only (`blockIdx`)
  - does not declare that the loop is, in fact, parallel (use independent if needed)
- `!$acc loop gang(32)`
  - runs in 'parallel' mode only with `gridDim` == 32 (32 blocks)
- `!$acc loop vector(128)`
  - runs in 'vector' mode (`threadIdx`) with `blockDim` == 128
    - vector size, if present, must be compile-time constant
- `!$acc loop gang vector(128)`
  - strip mines loop
  - inner loop runs in vector mode, 128 threads (`threadIdx`)
  - outer loop runs across thread blocks (`blockIdx`)

Data Region

- C

```
#pragma acc data
{
     ....
}
```

- Fortran

```
!$acc data
     ....
!$acc end data
```

- May span across host code and multiple compute regions
- May be nested
  - May not be nested within a compute region
- <u>Data is not implicitly synchronized between the host and device</u>

Data Clauses

- Data clauses
  - `copy( list )`
  - `copyin( list )`
  - `copyout( list )`
  - `create( list )`
  - `present( list )`
  - `present_or_copy(list)` `pcopy(list)`
  - `present_or_copyin(list)` `pcopyin(list)`
  - `present_or_copyout(list)` `pcopyout(list)`
  - `present_or_create(list)` `pcreate(list)`
  - `deviceptr( list )`

OpenACC Conditional Data Clauses

- OpenACC specification defines a set of conditional data clauses that only perform memory allocation or movement when the data is not already present on the device.
- **These conditional data clauses start with `present_or_`, which can be abbreviated with a "`p`" at the beginning of the clause.**
- `present_or_copy(list) or pcopy(list)` allocate and copy if needed and **copy back to host on exit**
- `present_or_copyin(list) or pcopyin(list)` allocate and copy if needed (no copy back)
- `present_or_copyout(list) or pcopyout(list)` allocate if not present and copy back to host on exit
- `present_or_create(list) or pcreate(list)` allocate if not present on device

Some hints for conditional data clause use

- `present`
  - This is for variables that have been copyin, copy or created up the call chain
    - If you forget this, you could be creating an error: Compiler will copy in the host version when you are expecting the device version.
- `present_or_create`
  - This is for variables that are only going to be used on the device.
- `present_or_copyin`
  - This is for variables that must be copied in from the host; however, they do not change after the first copyin.

Data Regions Across Procedures

```
subroutine sub( a, b )
 real :: a(:), b(:)
  !$acc kernels pcopyin(b)
    do i = 1,n
      a(i) = a(i) * b(i)
    enddo
  !$acc end kernels
  ...
end subroutine

subroutine bus(x, y)
 real :: x(:), y(:)
 !$acc data copy(x)
 call sub( x, y )
 !$acc end data
```

Update Directives

- `update host( list )`
- `update device( list )`
  - data must be in a data allocate clause for an enclosing data region
  - both may be on a single line
    - `update host(list) device(list)`
- Data update clauses on data construct (PGI)
  - `updatein( list )` or `update device( list )`
  - `updateout( list )` or `update host ( list )`
  - shorthand for update directive just inside data construct

Porting SEISMIC CPML

- Set of ten open-source Fortran90 programs
- Solves two-dimensional or three-dimensional isotropic or anisotropic elastic, viscoelastic or poroelastic wave equation.
- Uses finite-difference method with Convolutional or Auxiliary Perfectly Matched Layer (C-PML or ADE-PML) conditions
- Developed by Dimitri Komatitsch and Roland Martin from University of Pau, France.
- Accelerated source used is taken from the 3D elastic finite-difference code in velocity and stress formulation with Convolutional-PML (C-PML) absorbing conditions.

http://www.pgroup.com/lit/articles/insider/v4n1a3.htm

Step 1: Evaluation

- ~~Is my algorithm right for a GPU?~~
- My restatement – "What parts of my code are most amenable to computation on the GPU?"
  - SEISMIC_CPML models seismic waves through the earth. Has an outer time step loop with 9 inner parallel loops. Uses MPI and OpenMP parallelization.
- ~~Good candidate for the GPU, but not ideal.~~

Step 2: Add Compute Regions

```
!$acc kernels
 do k = kmin,kmax
  do j = NPOINTS_PML+1, NY-NPOINTS_PML
   do i = NPOINTS_PML+1, NX-NPOINTS_PML
   total_energy_kinetic = total_energy_kinetic + 0.5d0 * rho*(vx(i,j,k)**2 + vy(i,j,k)**2 + vz(i,j,k)**2)
   epsilon_xx = ((lambda + 2.d0*mu) * sigmaxx(i,j,k) - lambda * sigmayy(i,j,k) - lambda*sigmazz(i,j,k)) / (4.d0 * mu * (lambda + mu))
   epsilon_yy = ((lambda + 2.d0*mu) * sigmayy(i,j,k) - lambda * sigmaxx(i,j,k) - lambda*sigmazz(i,j,k)) / (4.d0 * mu * (lambda + mu))
   epsilon_zz = ((lambda + 2.d0*mu) * sigmazz(i,j,k) - lambda * sigmaxx(i,j,k) - lambda*sigmayy(i,j,k)) / (4.d0 * mu * (lambda + mu))
   epsilon_xy = sigmaxy(i,j,k) / (2.d0 * mu)
   epsilon_xz = sigmaxz(i,j,k) / (2.d0 * mu)
   epsilon_yz = sigmayz(i,j,k) / (2.d0 * mu)
   total_energy_potential = total_energy_potential + 0.5d0 * (epsilon_xx * sigmaxx(i,j,k) + epsilon_yy * sigmayy(i,j,k) + &
     epsilon_yy * sigmayy(i,j,k)+ 2.d0 * epsilon_xy * sigmaxy(i,j,k) + 2.d0*epsilon_xz * sigmaxz(i,j,k)+2.d0*epsilon_yz * sigmayz(i,j,k))
   enddo
  enddo
 enddo
!$acc end kernels
```

Compiler Feedback

```
% pgfortran -Mmpi=mpich2 -fast -acc -Minfo=accel
seismic_CPML_3D_isotropic_MPI_OACC_1.F90 -o gpu1.out
seismic_cpml_3d_iso_mpi_openmp:
   1107, Generating copyin(vz(11:91,11:631,kmin:kmax))
         Generating copyin(vy(11:91,11:631,kmin:kmax))
         Generating copyin(vx(11:91,11:631,kmin:kmax))
         Generating copyin(sigmaxx(11:91,11:631,kmin:kmax))
         Generating copyin(sigmayy(11:91,11:631,kmin:kmax))
         Generating copyin(sigmazz(11:91,11:631,kmin:kmax))
         Generating copyin(sigmaxy(11:91,11:631,kmin:kmax))
         Generating copyin(sigmaxz(11:91,11:631,kmin:kmax))
         Generating copyin(sigmayz(11:91,11:631,kmin:kmax))
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
```

```
1108, Loop is parallelizable
1109, Loop is parallelizable
1110, Loop is parallelizable
      Accelerator kernel generated
    1108, !$acc do gang, vector(4)
    1109, !$acc do gang, vector(4)
    1110, !$acc do vector(16)
1116, Sum reduction generated for total_energy_kinetic
1134, Sum reduction generated for total_energy_potential
```

Initial Timings

Version	MPI Processes	OpenMP Threads	GPUs	Time (sec)
Original MPI/OMP	2	4	0	951
ACC Step 1	2	0	2	3031

System Info:
4 Core Intel Core-i7 920 Running at 2.67Ghz
Includes 2 Tesla C2070 GPUs
Problem Size: 101x641x128

Why the slowdown?

Step 3: Optimize Data Movement

```
!$acc data                                                                  &
!$acc      copyin(a_x_half,b_x_half,k_x_half,                               &
!$acc                a_y_half,b_y_half,k_y_half,                            &
!$acc                a_z_half,b_z_half,k_z_half,                            &
!$acc                a_x,a_y,a_z,b_x,b_y,b_z,k_x,k_y,k_z,                   &
!$acc                sigmaxx,sigmaxz,sigmaxy,sigmayy,sigmayz,sigmazz,       &
!$acc                memory_dvx_dx,memory_dvy_dx,memory_dvz_dx,             &
!$acc                memory_dvx_dy,memory_dvy_dy,memory_dvz_dy,             &
!$acc                memory_dvx_dz,memory_dvy_dz,memory_dvz_dz,             &
!$acc                memory_dsigmaxx_dx, memory_dsigmaxy_dy,                &
!$acc                memory_dsigmaxz_dz, memory_dsigmaxy_dx,                &
!$acc                memory_dsigmaxz_dx, memory_dsigmayz_dy,                &
!$acc                memory_dsigmayy_dy, memory_dsigmayz_dz,                &
!$acc                memory_dsigmazz_dz)

  do it = 1,NSTEP
.... Cut ....
  enddo
!$acc end data
```

Timings Continued

Version	MPI Processes	OpenMP Threads	GPUs	Time (sec)
Original MPI/OMP	2	4	0	951
ACC Step 1	2	0	2	3031
ACC Step 2	2	0	2	124

System Info:
4 Core Intel Core-i7 920 Running at 2.67Ghz
Includes 2 Tesla C2070 GPUs

Data movement time now only 5 seconds!

Step 4: Fine Tune Schedule

```
!$acc loop vector(4)
  do k=k2begin,NZ_LOCAL
   kglobal = k + offset_k
!$acc loop gang, vector(4)
   do j=2,NY
!$acc loop gang, vector(16)
     do i=1,NX-1
      value_dvx_dx = (vx(i+1,j,k)-vx(i,j,k)) * ONE_OVER_DELTAX
      value_dvy_dy = (vy(i,j,k)-vy(i,j-1,k)) * ONE_OVER_DELTAY
      value_dvz_dz = (vz(i,j,k)-vz(i,j,k-1)) * ONE_OVER_DELTAZ
```

Final Timings

Version	MPI Processes	OpenMP Threads	GPUs	Time (sec)	Approx. Programming Time (min)
Original MPI/OMP	2	4	0	951	
ACC Step 1	2	0	2	3031	10
ACC Step 2	2	0	2	124	180
ACC Step 3	2	0	2	120	120

7x in 5 Hours!

OpenACC is not an Island

- OpenACC allows very high level expression of parallelism and data movement.
- It's still possible to leverage low-level approaches such as CUDA C, CUDA Fortran, and GPU Libraries.

CUDA C Primer

Standard C

```
void saxpy(int n, float a,
           float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}


int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

Parallel C

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

- Serial loop over 1M elements, executes 1M times sequentially.
- Data is resident on CPU.

- Parallel kernel, executes 1M times in parallel in groups of 256 elements.
- Data must be copied to/from GPU.

http://developer.nvidia.com/cuda-toolkit

CUDA C Interoperability

OpenACC Main

```fortran
program main
  integer, parameter :: N = 2**20
  real, dimension(N) :: X, Y
  real               :: A = 2.0

  !$acc data
  ! Initialize X and Y
  ...

  !$acc host_data use_device(x,y)
  call saxpy(n, a, x, y)
  !$acc end host_data
  !$acc end data

end program
```

CUDA C Kernel & Wrapper

```chapel
__global__
void saxpy_kernel(int n, float a,
                  float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

void saxpy(int n, float a, float *dx, float *dy)
{
  // Launch CUDA Kernel
  saxpy_kernel<<<4096,256>>>(N, 2.0, dx, dy);
}
```

- It's possible to interoperate from C/C++ or Fortran.
- OpenACC manages the data and passes device pointers to CUDA.

- CUDA kernel launch wrapped in function expecting device arrays.
- Kernel is launch with arrays passed from OpenACC in main.

CUDA C Interoperability (Reversed)

OpenACC Kernels

```
void saxpy(int n, float a, float *
restrict x, float * restrict y)
{
  #pragma acc kernels
deviceptr(x[0:n],y[0:n])
  {
    for(int i=0; i<n; i++)
    {
      y[i] += 2.0*x[i];
    }
  }
}
```

By passing a device pointer to an OpenACC region, it's possible to add OpenACC to an existing CUDA code.

CUDA C Main

```
int main(int argc, char **argv)
{
  float *x, *y, tmp;
  int n = 1<<20, i;

  cudaMalloc((void*)&x,(size_t)n*sizeof(float));
  cudaMalloc((void*)&y,(size_t)n*sizeof(float));

  ...

  saxpy(n, 2.0, x, y);
  cudaMemcpy(&tmp,y,(size_t)sizeof(float),
             cudaMemcpyDeviceToHost);
  return 0;
}
```

Memory is managed via standard CUDA calls.

CUDA Fortran

Standard Fortran

```
module mymodule contains
  subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    do i=1,n
      y(i) = a*x(i)+y(i)
    enddo
  end subroutine saxpy
end module mymodule

program main
  use mymodule
  real :: x(2**20), y(2**20)
  x = 1.0, y = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy(2**20, 2.0, x, y)

end program main
```

- Serial loop over 1M elements, executes 1M times sequentially.
- Data is resident on CPU.

Parallel Fortran

```
module mymodule contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0, y_d = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)

end program main
```

- Parallel kernel, executes 1M times in parallel in groups of 256 elements.
- Data must be copied to/from GPU (implicit).

http://developer.nvidia.com/cuda-fortran

CUDA Fortran Interoperability

OpenACC Main

```
program main
  use mymodule
  integer, parameter :: N =
2**20
  real, dimension(N) :: X, Y

  X(:) = 1.0
  Y(:) = 0.0

  !$acc data copy(y) copyin(x)
  call saxpy(N, 2.0, x, y)
  !$acc end data

end program
```

CUDA Fortran Kernel & Launcher

```
module mymodule
  contains
  attributes(global) &
  subroutine saxpy_kernel(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy_kernel
  subroutine saxpy (n, a, x, y)
    use cudafor
    real, device :: x(:), y(:)
    real :: a
    integer :: n
    call saxpy_kernel<<<4096,256>>>(n, a, x, y)
  end subroutine saxpy
end module mymodule
```

- Thanks to the “device” attribute in saxpy, no host_data is needed.
- OpenACC manages the data and passes device pointers to CUDA.

- CUDA kernel launch wrapped in function expecting device arrays.
- Kernel is launch with arrays passed from OpenACC in main.

OpenACC with CUDA Fortran Main

Using the "deviceptr" data clause makes it possible to integrate OpenACC into an existing CUDA application.

CUDA C takes a few more tricks to compile, but can be done.

In theory, it should be possible to do the same with C/C++ (including Thrust), but in practice compiler incompatibilities make this difficult.

CUDA Fortran Main w/ OpenAcc Region

```
program main
  use cudafor
  integer, parameter :: N = 2**20
  real, device, dimension(N) :: X, Y
  integer :: i
  real :: tmp

  X(:) = 1.0
  Y(:) = 0.0

  !$acc kernels deviceptr(x,y)
  y(:) = y(:) + 2.0*x(:)
  !$acc end kernels

  tmp = y(1)
  print *, tmp
end program
```

CUBLAS Library

Serial BLAS Code

```
int N = 1<<20;


...

// Use your choice of blas library

// Perform SAXPY on 1M elements
blas_saxpy(N, 2.0, x, 1, y, 1);
```

Parallel cuBLAS Code

```
int N = 1<<20;


cublasInit();
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasShutdown();
```

You can also call cuBLAS from Fortran, C++, Python, and other languages

http://developer.nvidia.com/cublas

CUBLAS Library & OpenACC

OpenACC can interface with existing GPU-optimized libraries (from C/C++ or Fortran).

This includes…

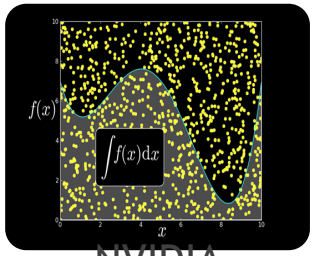
- CUBLAS
- Libsci_acc
- CUFFT
- MAGMA
- CULA
- …

OpenACC Main Calling CUBLAS

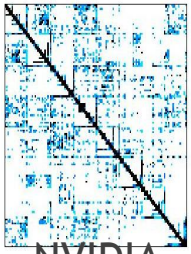
```
int N = 1<<20;
float *x, *y
// Allocate & Initialize X & Y
...

cublasInit();

#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
  #pragma acc host_data use_device(x,y)
  {
    // Perform SAXPY on 1M elements
    cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
  }
}

cublasShutdown();
```

Some GPU-accelerated Libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

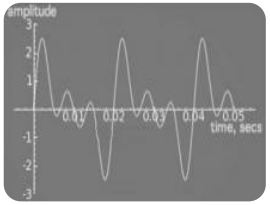
CULA tools

GPU Accelerated Linear Algebra

MAGMA

Matrix Algebra on GPU and Multicore

open source initiative



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

ArrayFire Matrix Computations

CUSP

Sparse Linear Algebra

open source initiative

Thrust

C++ STL Features for CUDA

open source initiative

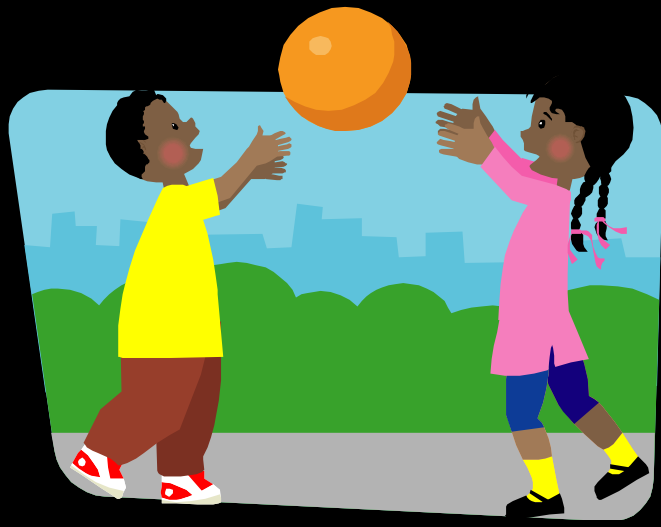
Interoperating with accelerated libraries -- Cray LibSci

```
!$acc data copy(a,b,c)
!$acc parallel

!Do Something

!$acc end parallel

!$acc host_data use_device(a,b,c)

call dgemm_acc('n','n',m,n,k,&
               alpha,a,lda,&
               b,ldb,beta,c,ldc)
!$acc end host_data
!$acc end data
```

How to play well with others

My advice is to do the following:

1. Start with OpenACC
   - Expose high-level parallelism
   - Ensure correctness
   - Optimize away data movement last
2. Leverage other work that's available (even if it's not OpenACC)
   - Common libraries (good software engineering practice)
   - Lots of CUDA already exists
3. Share your experiences
   - OpenACC is still very new, best practices are still forming.
   - Allow others to leverage your work.



More Interoperability Advice

- OpenACC provides a very straightforward way to manage data structures without needing 2 pointers (host & device), so use it at the top level.
- CUDA provides very close-to-the-metal control, so it can be used for very highly tuned kernels that may be called from OpenACC
- Compilers do complex tasks such as reductions very well, so let them.

Interoperating with OpenMP and MPI

- Develop a single source code that implements OpenMP and OpenACC in such a way that application can be efficiently run on:
  - Multi-core MPP systems
  - Multi-core MPP systems with companion accelerator
    - Nvida
    - Intel
    - AMD
    - Whatever
- Clearly identify three levels of parallelism
  - MPI/PGAS between NUMA/UMA memory regions
  - Threading within the NUMA/UMA memory region
    - How this is implemented is important – OpenMP/OpenACC is most portable
  - SIMDization at a low level
    - How this is coded is important – compilers have different capability
- We do want a performance/portable application at the end

Relationship between OpenMP and OpenACC

```
DO WHILE TIME < MAXIMUM TIME
  !$omp parallel default(shared) private(…….)
            call crunch0 (Contains OpenMP  )
    !$omp do

            call crunch1
            call crunch2

    !$omp end do
            call communicate0 (Contains MPI)

     !$omp do
    call crunch3 (Contains OpenMP  )
  !$omp end parallel

END DO
```

Relationship between OpenMP and OpenACC

```
!$acc data copyin(OpenMP shared data…
!$acc  present_or_create( OpenMP private data…
DO WHILE TIME < MAXIMUM TIME
  !$omp parallel default(shared)  private(…….)
            call crunch0 (Contains OpenMP and OpenACC)
    !$omp do
    !$acc parallel loop
            call crunch1
            call crunch2
    !$acc parallel loop
    !$omp end do
            call communicate0 (Contains MPI)

     !$omp do
    call crunch3 (Contains OpenMP and OpenACC)
  !$omp end parallel

END DO
!$acc end data
```

Relationship between OpenMP and OpenACC

```
DO WHILE TIME < MAXIMUM TIME
  !$omp parallel do default(shared) private(…….)
    DO K = 1, KMAX
        call crunch0

        DO J = 1, JMAX
            call crunch1
            call crunch2
        END DO

    call communicate0 (Contains MPI)
    call crunch3
    END DO
  !$omp end parallel do
END DO
```

Relationship between OpenMP and OpenACC

```
!$acc data copyin(OpenMP shared data…
!$acc  present_or_create( OpenMP private data…
DO WHILE TIME < MAXIMUM TIME
  !$omp parallel do default(shared) private(…….)
     DO K = 1, KMAX
          call crunch0(Contains OpenACC with ASYNC(K))
   !$ acc parallel loop present(….)  ASYNC(K)
          DO J = 1, JMAX
               call crunch1
               call crunch2
          END DO
   !$acc end parallel loop
   !$acc wait(K)
     call communicate0 (Contains MPI)
     call crunch3   (Contains OpenACC with ASYNC(K) )
  !$omp end parallel loop
END DO
```

OpenACC 2.0 - Highlights

- Procedure calls, separate compilation
- Nested parallelism (support for Dynamic Parallelism)
- Loop tile clause
- Data management features and global data
- Device-specific tuning
- Asynchronous behavior additions
- New API routines
- New atomic contstruct
- New default(none) data clause

Procedure calls – OpenACC 1.0

```
#pragma acc parallel loop
for ( int i=0; i<n; ++i) {
  foo(v,i);
  //must inline foo
}
```

Procedure calls – OpenACC 2.0

```
#pragma acc routine worker
extern void foo(float* v, int i);

#pragma acc parallel loop
for ( int i=0; i<n; ++i) {
  foo(v,i);
  //call on the device
}
```

Tell the compiler the level of parallelism in foo

Procedure calls – OpenACC 2.0

```
#pragma acc routine worker
extern void foo(float* v,int i);

#pragma acc parallel loop
for ( int i=0; i<n; ++i) {
  foo(v,i);
  //call on the device
}
```

```
#pragma acc routine worker
void foo(float* v, int i) {
  #pragma acc loop worker
  for ( int j=0; j<n; ++j) {
    v[i*n+j] = 1.0f/(i*j);
  }
}
```

Nested Parallelism

```
#pragma acc routine
extern void foo(float* v,int i);

#pragma acc parallel loop
for ( int i=0; i<n; ++i) {
  foo(v,i);
  //call on the device and spawn new
  //threads
}
```

```
#pragma acc routine
void foo(float* v, int i) {
  #pragma acc parallel loop
  for ( int j=0; j<n; ++j) {
    v[i*n+j] = 1.0f/(i*j);
  }
}
```

Loop `tile` Clause

Block loops into 8x8 tiles.

- OpenACC 1.0 does not provide a standard way to decompose loops into 2D threadblocks
  - May better exploit data locality

```
#pragma acc loop tile(8,8)
for ( int i=0; i<n; ++i)
{
  for ( int j=0; j<n; ++j)
  {
    v[i*n+j] = 1.0f/(i*j);
  }
}
```

Device-specific tuning – device_type

```
#pragma acc routine worker
extern void foo(float* v,int i);

#pragma acc parallel loop \
  num_workers(384)

for ( int i=0; i<n; ++i) {
  foo(v,i);
}
```

```
#pragma acc routine worker
extern void foo(float* v,int i);

#pragma acc parallel loop \
  device_type(nvidia) num_workers(256)\
  device_type(radeon) num_workers(512)
for ( int i=0; i<n; ++i) {
  foo(v,i);
}
```

OpenACC 1.0 - `async` clause

```
#pragma acc parallel async(1)
{… /*kernel A*/}
do_something_on_host()
#pragma acc parallel async(2)
{…/*Kernel B*/}
#pragma acc parallel async(2)
{…/*Kernel C*/}
```

The **async** clause is optional on the **parallel** and **kernels** constructs; when there is no **async** clause, the host process will wait until the parallel or kernels region is complete before executing any of the code that follows the construct. When there is an **async** clause,

Do not wait for kernel completion

Executes concurrently with kernels

(potentially) executes concurrently with kernels

OpenACC 1.0 – `wait` directive

```
#pragma acc parallel async(1)
{… /*kernel A*/}
do_something_on_host()
#pragma acc parallel async(2)
{…/*Kernel B*/}
#pragma acc parallel async(2)
{…/*Kernel C*/}

#pragma acc wait(1)
#pragma acc parallel async(2)
{…}
```

Wait for kernel A in queue (stream) 1 - blocks host

Schedule new work in queue (stream) 2 that depends on queue 1.

OpenACC 2.0 – `wait` directive with `async` clause

```
#pragma acc parallel async(1)
{… /*kernel A*/}
do_something_on_host()
#pragma acc parallel async(2)
{…/*Kernel B*/}
#pragma acc parallel async(2)
{…/*Kernel C*/}

#pragma acc wait(1)
#pragma acc parallel async(2)
{…}
```

```
#pragma acc parallel async(1)
{… /*kernel A*/}
do_something_on_host()
#pragma acc parallel async(2)
{…/*Kernel B*/}
#pragma acc parallel async(2)
{…/*Kernel C*/}

#pragma acc wait(1) async(2)
#pragma acc parallel async(2)
{…}
```

New API routines

- Improved Data management for C/C++
  - acc_copyin
  - acc_present_or_copyin
  - acc_create
  - acc_present_or_create
  - acc_copyout
  - acc_delete
  - acc_map_data
  - acc_unmap_data
  - acc_deviceptr
  - acc_hostptr
  - acc_is_present
  - acc_memcpy_to_device
  - acc_memcpy_from_device
  - acc_update_device
  - acc_update_self
- Expanded Interoperability with CUDA, OpenCL, & Xeon Phi
  - acc_get_cuda_stream
  - acc_get_current_cuda_device
  - …

Hands-On Examples

https://github.com/olcf/OpenACC_workshop_072013

VH1_example and himeno example