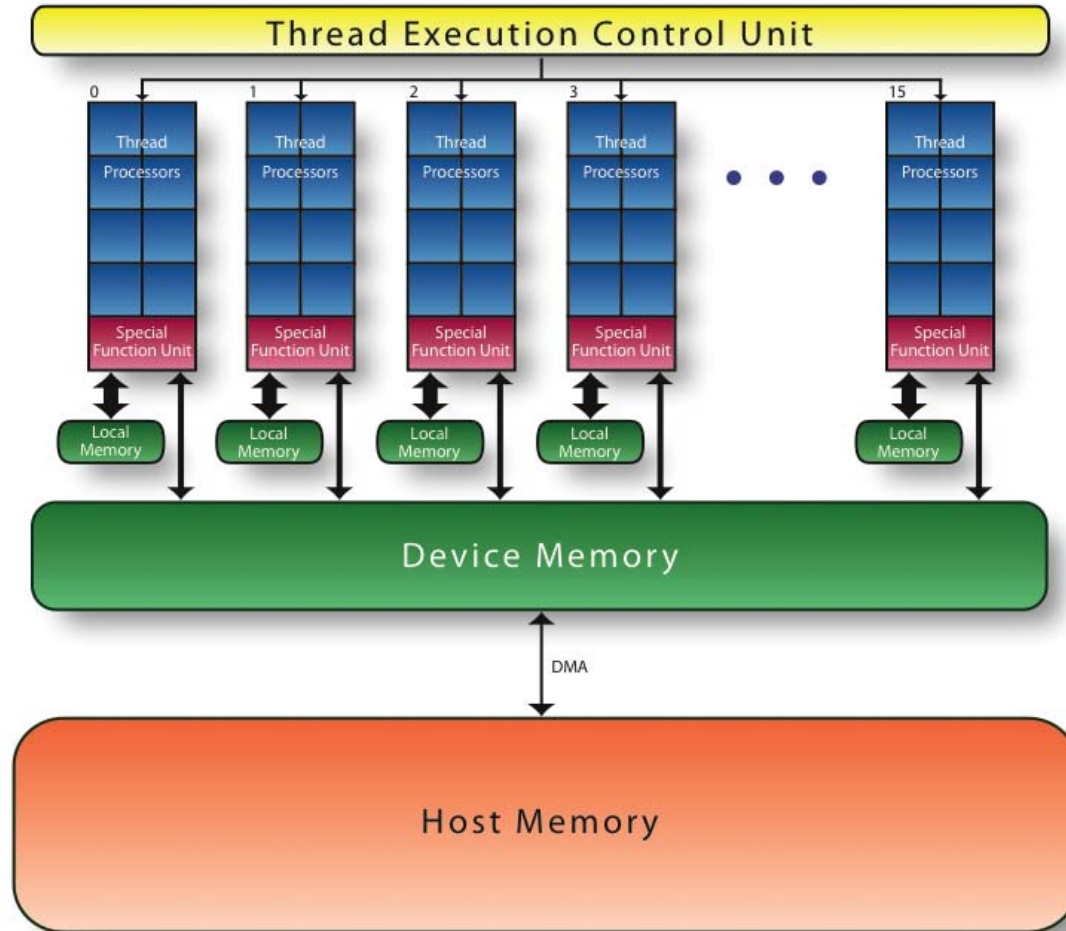


PGI Compilers, GPUs and You!



Abstracted GPU/Accelerator Architecture



OpenMP Programming Model

Parallel Matrix Multiply in OpenMP

```
!$omp parallel
!$omp do, schedule(static), shared(a,b,c)
  do j = 1,m
    do i = 1,n
      a(i,j) = 0.0
    enddo
    do k = 1,p
      do i = 1,n
        a(i,j) = a(i,j) + b(i,k) * c(k,j)
      enddo
    enddo
  enddo
!$omp end parallel
```



CUDA GPU Programming Model

CUDA host-side code for Matrix Multiply

```
cuInit(0);
cuDeviceGet( &device, devnum );
cuCtxCreate( &context, 0, device );
cuModuleLoad( &module, binfile );
cuModuleGetFunction( &func, module, "mmkernel" );
cuMemAlloc( &bp, memsize ); cuMemAlloc( &ap, memsize );
cuMemAlloc( &cp, memsize );
cuMemcpyHtoD( bp, b, memsize ); cuMemcpyHtoD( cp, c, memsize );
cuMemcpyHtoD( ap, a, memsize );
dim3 threads( 128 );
dim3 blocks( N/128, M );
func<<<blocks,threads>>>( ap, bp, cp, nsize, nsize, nsize,
                          matsize, matsize, matsize );

cuCtxSynchronize();
cuMemcpyDtoH( a, ap, memsize );
cuMemFree( ap );
cuMemFree( bp );
cuMemFree( cp );
cuModuleUnload( module );
cuCtxDestroy( context );
```



CUDA GPU Programming Model

CUDA GPU-side kernel for Matrix Multiply

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c,
          int pitch_a, int pitch_b, int pitch_c,
          int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*128 + tx;
    int j = blockIdx.y;
    __shared__ float cb0[128];
    float sum0 = 0.0;
    for( int ks = 0; ks < p; ks += 128 ){
        cb0[tx] = c[ks+tx+pitch_c*j];
        __syncthreads();
        for( int k = 0; k < 128; k++ ){
            sum0 += b[i+pitch_b*(k+ks)] * cb0[k];
        }
        __syncthreads();
    }
    a[i+pitch_a*j] = sum0;
}
```



PGI Accelerator Programming Model

Directive-based Matrix Multiply for x64+GPU target

```
!$acc region
  do j = 1,m
    do i = 1,n
      a(i,j) = 0.0
    enddo
    do k = 1,p
      do i = 1,n
        a(i,j) = a(i,j) + b(i,k) * c(k,j)
      enddo
    enddo
  enddo
!$acc end region
```



PGI Accelerator Compiler Feedback

Compiler Communicates to User how Kernel
was Mapped to the Accelerator

mm1:

8, Loop is parallelizable

9, Loop is parallelizable

Kernel schedule is 8(parallel), 9(parallel), 8(vector(16)),
9(vector(16))

12, Loop carried reuse of a prevents parallelization

13, Loop is parallelizable

Kernel schedule is 8(parallel), 13(parallel), 12(strip),
8(vector(16)), 13(vector(16)), 12(seq(16))



Same Basic Model for C - pragmas

```
#pragma acc region
{
    for(int opt = 0; opt < optN; opt++){
        float S = h_StockPrice[opt],
              X = h_OptionStrike[opt],
              T = h_OptionYears[opt];
        float sqrtT = sqrtf(T);
        float d1 = (logf(S/X) +
                   (Riskfree + 0.5 * Volatility * Volatility) * T)
                  / (Volatility * sqrtT);
        float d2 = d1 - Volatility * sqrtT;
        float cndd1 = CND(d1);
        float cndd2 = CND(d2);
        float expRT = expf(- Riskfree * T);
        h_CallResult[opt] = (S*cndd1-X*expRT*cndd2);
        h_PutResult[opt] = (X*expRT*(1.0-cndd2)-S*(1.0-cndd1));
    }
}
```



Accelerator Region Optional Clauses

Clauses Enable User-assisted Optimization
of Data Movement

```
!$acc region, copyin(b,c), copyout(a)
  do j = 1,m
    do i = 1,n
      a(i,j) = 0.0
    enddo
    do k = 1,p
      do i = 1,n
        a(i,j) = a(i,j) + b(i,k) * c(k,j)
      enddo
    enddo
  enddo
!$acc end region
```



Accelerator Region Optional Directives

DOALL-style Directives for User-assisted Optimization
of Loop Mapping across Asynchronous Dimension

```
!$acc region, copyin(b,c), copyout(a)
  !$acc do parallel
    do j = 1,m
      do i = 1,n
        a(i,j) = 0.0
      enddo
    do k = 1,p
      do i = 1,n
        a(i,j) = a(i,j) + b(i,k) * c(k,j)
      enddo
    enddo
  enddo
!$acc end region
```



Accelerator Region Optional Directives

Clauses for User-assisted Optimization Using Mixed-mode Execution

```
!$acc region, copyin(b,c), copyout(a)
  !$acc do parallel
    do j = 1,m
      !$acc do parallel,vector(256)
        do i = 1,n
          a(i,j) = 0.0
        enddo
      do k = 1,p
        !$acc do parallel,vector(256)
          do i = 1,n
            a(i,j) = a(i,j) + b(i,k) * c(k,j)
          enddo
        enddo
      enddo
    enddo
  !$acc end region
```



Accelerator Region Optional Directives

SIMD-style Directives/Clauses for User-assisted Optimization of Loops along Synchronous Dimension

```
!$acc region, copyin(b,c), copyout(a)
  !$acc do parallel
    do j = 1,m
      !$acc do vector(256), shortloop
        do i = 1,n
          a(i,j) = 0.0
        enddo
      do k = 1,p
        !$acc do vector(256), shortloop, cache(b,c)
          do i = 1,n
            a(i,j) = a(i,j) + b(i,k) * c(k,j)
          enddo
        enddo
      enddo
    enddo
  !$acc end region
```



Required Compiler Analysis

- Live variable analysis
- Array region analysis
- In/Out/Local variable/array analysis
- Private arrays
- Dependence analysis
- Parallelism detection



Host-side Code Generation

- Accelerator device connection / initialization
- Download code module(s)
- Allocate data memory (including private data)
- Upload input data
- Launch kernel(s)
- Download results
- Deallocate memory
- Clean up

PGI Accelerator Compiler automatically generates the equivalent of host-side CUDA code in x64 assembly code



Accelerator Code Generation

- Determine loop schedule
 - parallel loop
 - vector loop
 - unrolled loop
 - sequential loop
 - host loop
- Standard CG techniques
 - GPU run-time reoptimization

PGI Accelerator Compiler auto-generates equivalent of GPU-side CUDA kernel from directive-delimited Accelerator Region

