An Accelerator Programming Model for Multicore

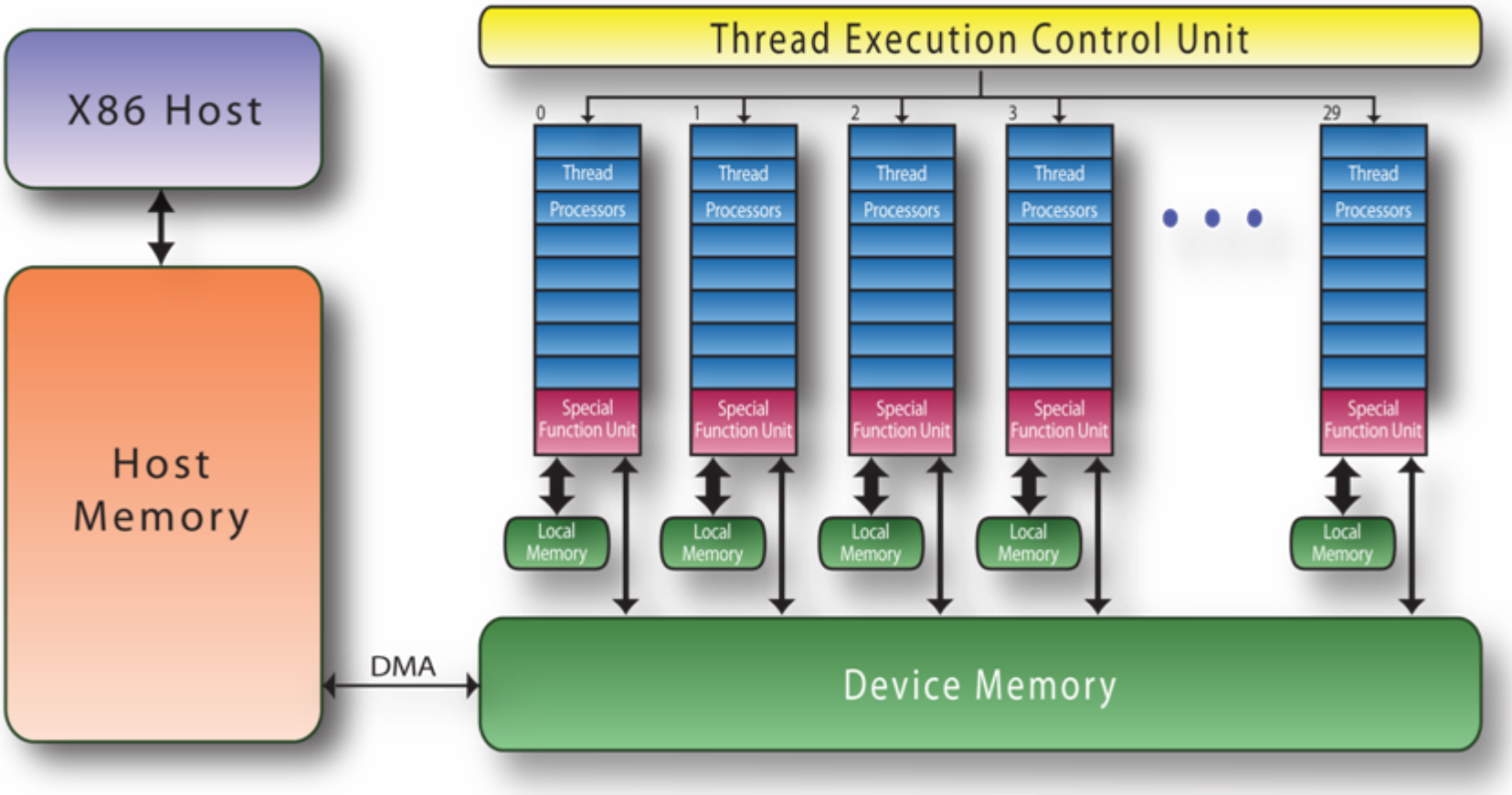
Brent Leback, Steven Nakamoto, and Michael Wolfe
PGI

May 6, 2009

Roadmap for this Talk

- Describe the problem of programming an x64+GPU system
- Describe the PGI Accelerator “Kernels” programming model
- Describe directives used to program to the model
- Show how the directives are compiled to an accelerator target
- Project how the directives could be applied to x64
- Discuss the generality and limitations of the model

Abstracted x64+Accelerator Architecture



PGI Accelerator Model Design

- **Based on two successful models: Vector Programming and OpenMP**
- **Vector Programming:**
  - No new language was needed
  - It was incremental
  - The compiler gave feedback
  - The programming model was portable
- **OpenMP**
  - Layered atop an underlying technology (pthreads in this case)
  - Directive-based, accessible for most developers
  - Sufficiently expressive to solve most problems
  - Code still works without the directives

Simple Fortran Matrix Multiply for an x64 Host, accelerated

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

Basic CUDA C Matrix Multiply Kernel for an NVIDIA GPU

```
extern "C" __global__ void
mmkernel( float* a,float* b,float* c,
          int la,int lb,int lc,int n,
          int m,int p )
{
    int i = blockIdx.x*64+threadIdx.x;
    int j = blockIdx.y;

    float sum = 0.0;
    for( int k = 0; k < p; ++k )
      sum += b[i+lb*k] * c[k+lc*j];
    a[i+la*j] = sum;
}
```

```
extern "C" __global__ void
mmkernel( float* a, float* b, float* c, int la, int lb, int lc, int n, int m, int p )
{
    int tx = threadIdx.x;
    int i = blockIdx.x*128 + tx;  int j = blockIdx.y*4;
    __shared__ float cb0[128], cb1[128], cb2[128], cb3[128];

    float sum0 = 0.0, sum1 = 0.0, sum2 = 0.0, sum3 = 0.0;
    for( int ks = 0; ks < p; ks += 128 ){
      cb0[tx] = c[ks+tx+lc*j];     cb1[tx] = c[ks+tx+lc*(j+1)];
      cb2[tx] = c[ks+tx+lc*(j+2)]; cb3[tx] = c[ks+tx+lc*(j+3)];
      __syncthreads();
      for( int k = 0; k < 128; k+=4 ){
        float rb = b[i+lb*(k+ks)];
         sum0 += rb * cb0[k];   sum1 += rb * cb1[k];
          sum2 += rb * cb2[k];   sum3 += rb * cb3[k];
        rb = b[i+lb*(k+ks+1)];
         sum0 += rb * cb0[k+1]; sum1 += rb * cb1[k+1];
          sum2 += rb * cb2[k+1]; sum3 += rb * cb3[k+1];
        rb = b[i+lb*(k+ks+2)];
         sum0 += rb * cb0[k+2]; sum1 += rb * cb1[k+2];
          sum2 += rb * cb2[k+2]; sum3 += rb * cb3[k+2];
        rb = b[i+lb*(k+ks+3)];
         sum0 += rb * cb0[k+3]; sum1 += rb * cb1[k+3];
          sum2 += rb * cb2[k+3]; sum3 += rb * cb3[k+3];
      }
      __syncthreads();
    }
    a[i+la*j] = sum0;     a[i+la*(j+1)] = sum1;
     a[i+la*(j+2)] = sum2; a[i+la*(j+3)] = sum3;
}
```

Optimized CUDA C Matrix Multiply Kernel

Host-side CUDA C Matrix Multiply GPU Control Code

```
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( matsize/128, matsize/4 );
mmkernel<<<blocks,threads>>>(ap,bp,cp,nsize,nsize,
                    nsize,matsize,matsize,matsize);

cuMemcpyDtoH( a, ap, memsize );
```

What is a “Kernels” Programming Model?

- **Kernel** – a multidimensional parallel loop, where the body of the loop is the kernel code and the loops define the index set or domain
- **Program** – a sequence of kernels, where each kernel executes to completion over its index set before the next kernel can start
- **Parallelism** is exploited between iterations of a kernel, not between multiple kernels executing in parallel

Kernels Model Pseudo-code for Simple Matrix Multiply

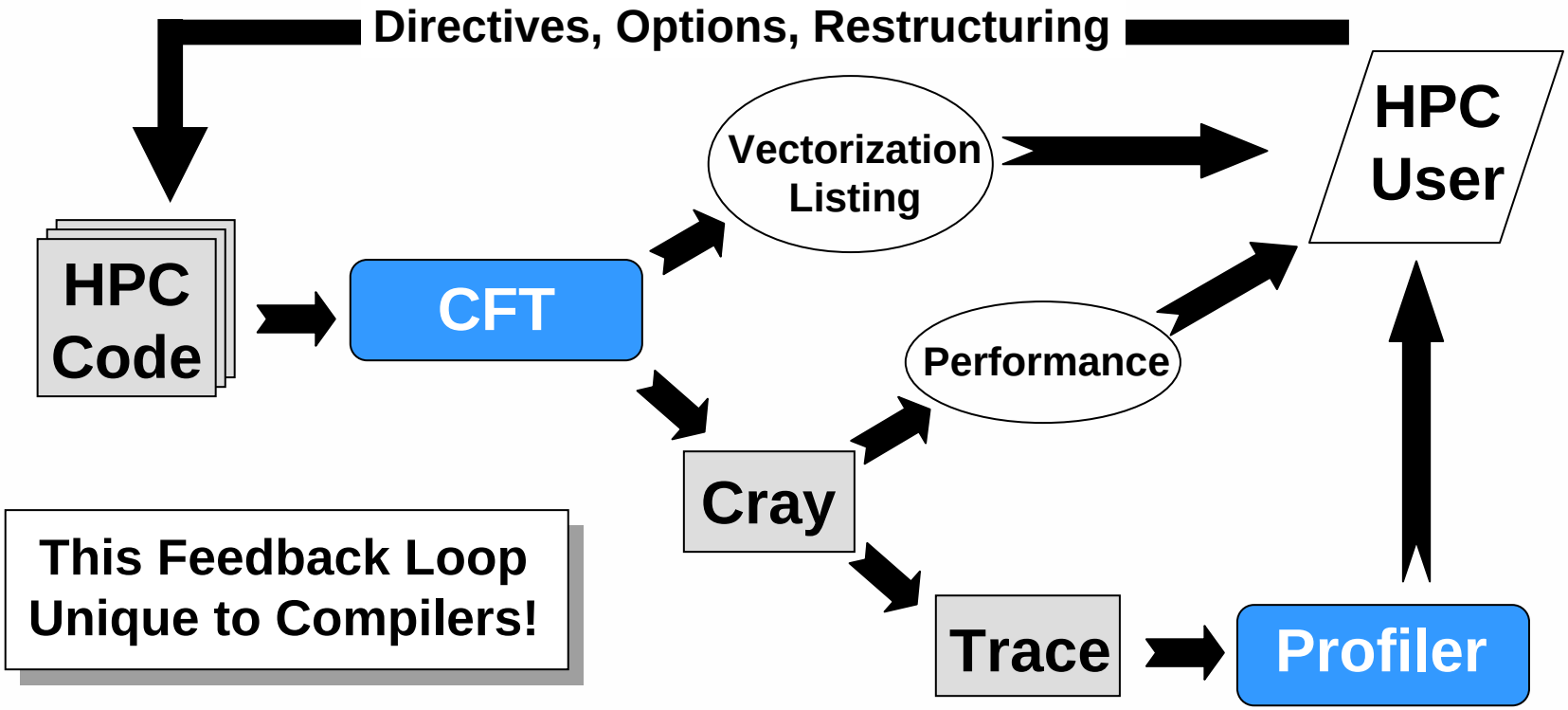
```
dopar j = 1, m
   dopar is = 1, n, 64
      dovec i = is, is+63
         sum = 0.0
         doseq k = 1, p
            sum += b(i,k) * c(k,j)
         enddo
         a(i,j) = sum
      enddo
   enddo
enddo
```

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));
    }
}
```

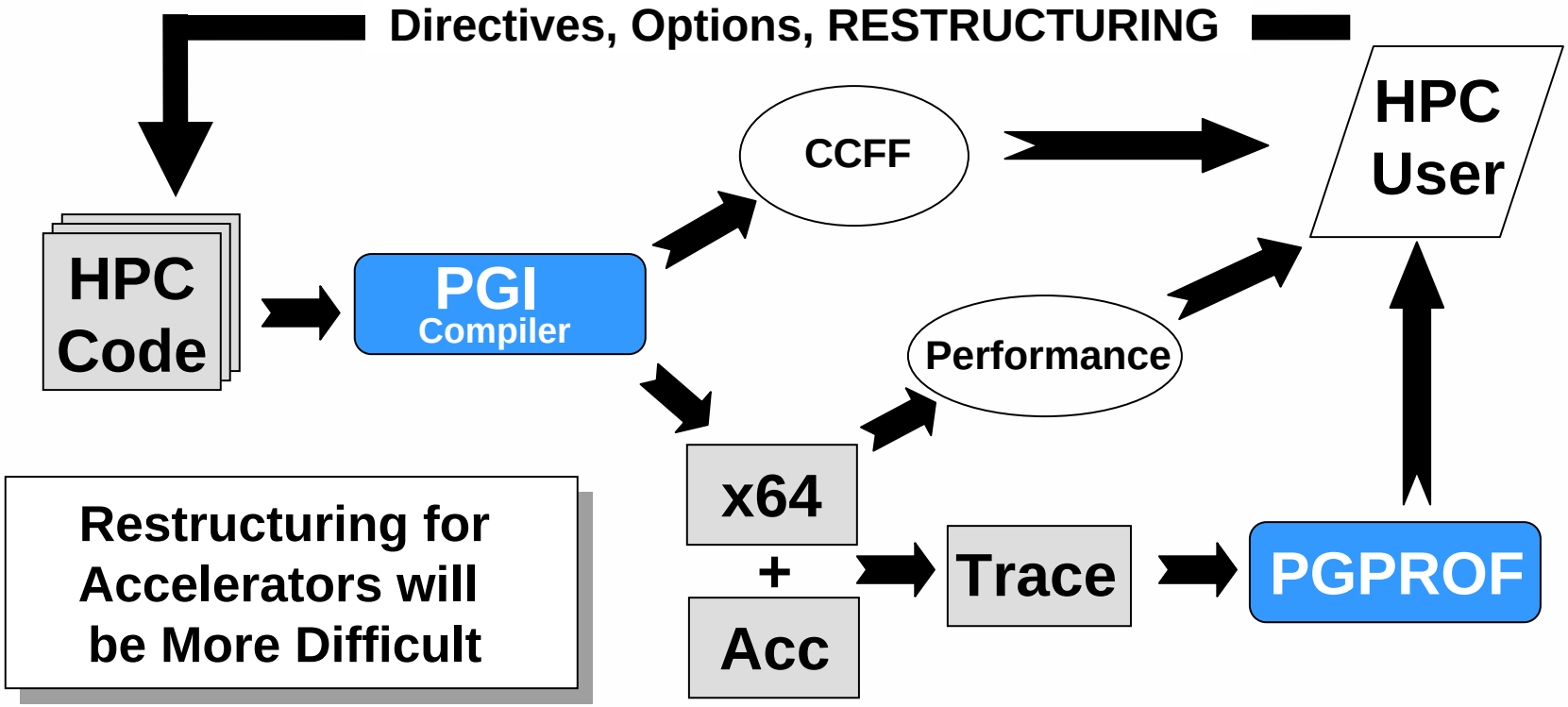
How did we make Vectors Work?

Compiler-to-Programmer Feedback – a classic "Virtuous Cycle"



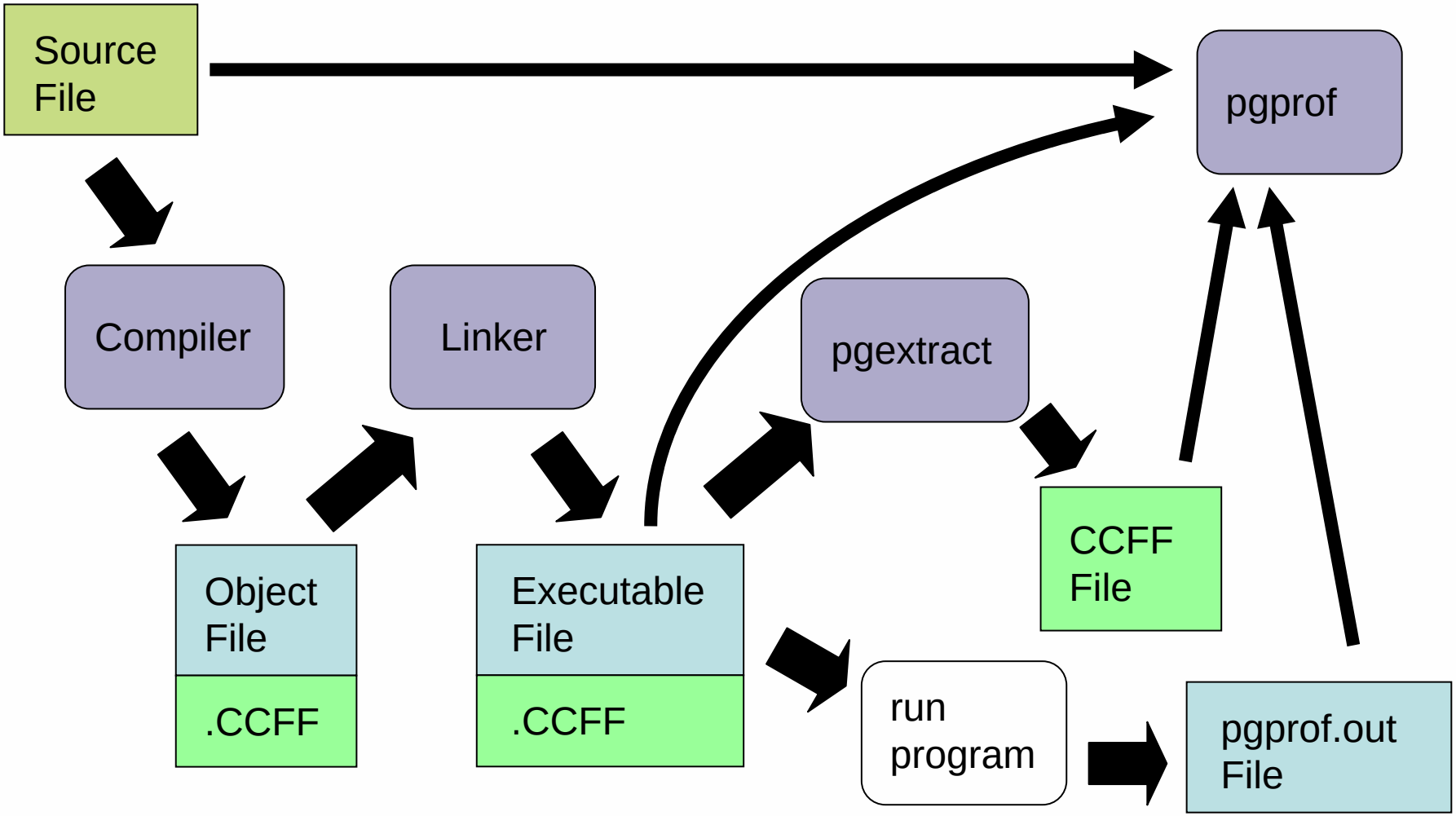
We can use this same methodology to enable effective migration of applications to Multi-core and Accelerators

Compiler-to-Programmer Feedback



Common Compiler Feedback Format

http://www.pgroup.com/resources/ccff.htm



Types of Compiler Feedback

- How the function was compiled
- Inter-procedural optimizations
- Profile-feedback runtime data
  - Block execution counts
  - Loop counts, range of counts
- Compiler optimizations, missed opportunities
  - Vectorization, parallelization
  - Altcode, re-ordering of loops, inlining
  - X64+GPU code generation, GPU kernel mapping, data movement
- Compute intensity – important for GPUs & Multi-core

Compiler-to-User Feedback

```
% pgf95 -fast -ta=nvidia -Minfo mm.F90
mm1:
      4, Generating copyin(c(1:m,1:m))
         Generating copyin(b(1:m,1:m))
         Generating copy(a(1:m,1:m))
      5, Loop is parallelizable
      6, Loop carried dependence of a prevents parallelization
         Loop carried backward dependence of a prevents vectorization
         Cached references to size [16x16] block of b
         Cached references to size [16x16] block of c
      7, Loop is parallelizable
         Kernel schedule is 5(parallel), 7(parallel), 6(strip),
             5(vector(16)), 7(vector(16)), 6(seq(16))
         Parallel read/write of a
```

Clauses for Tuning Data Movement

```
!$acc region copyin(b(1:n,1:p),c(1:p,1:m))
!$acc&       copy(a(1:n,1:m)) local(i,j,k)
      do j = 1, m
         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
```

Directives for Tuning Kernel Mapping

```
!$acc region copyin(b(1:n,1:p),c(1:p,1:m))
!$acc&       copy(a(1:n,1:m)) local(i,j,k)
!$acc do parallel
      do j = 1, m
         do k = 1, p
!$acc       do parallel, vector(64)
            do i = 1, n
               a(i,j) = a(i,j) + b(i,k)*c(k,j)
            enddo
         enddo
      enddo
!$acc end region
```

Directives for Tuning Data Locality

```
!$acc device data(t)
. . .
!$acc region
do j = 2,m-1
  do i = 2,n-1
    r(i-1,j-1) = 2.0 * t(i,j) - 0.25*(s(i-1,j) + &
                 s(i+1,j) + s(i,j-1) + s(i,j+1))
  enddo
enddo
!$acc end region
. . .
!$acc region
< another kernel that uses t >
!$acc end region
!$acc end device data region
```

Accelerator Directives Processing

- Live variable and array region analysis to augment information in region directives and determine in / out datasets for the region
- Dependence and parallelism analysis to augment information in loop directives and determine loops that can be executed in parallel
- Select mapping of loop(s) onto hardware parallelism, SIMD/vector and MIMD/parallel dimensions, strip mining and tiling for performance
- Extract the kernel or kernels, generate target code for each kernel
- Lots of opportunity for optimization of kernel code - loop unrolling, software data cache usage, register lifetime management (minimize register usage to maximize multithreading potential)
- Generate host code to drive and launch kernels, allocate GPU memory, copy data from host to GPU, copy results back to host, continue on host, generate host version of loop(s) OR, on multicore, tune for data locality, prefetching, caching, etc.
- Generate feedback to programmer

PGI Accelerator Model Advantages

- **Minimal changes to the language** – directives/pragmas, in the same vein as vector or OpenMP parallel directives
- **Minimal library calls** – usually none
- **Standard x64 toolchain** – no changes to makefiles, linkers, build process, standard libraries, other tools
- **Not a "platform"** – binaries will execute on any compatible x64+GPU hardware system
- **Performance feedback** – learn from and leverage the success of vectorizing compilers in the 1970s and 1980s
- **Incremental program migration** – put migration decisions in the hands of developers
- **PGI Unified Binary Technology** – ensures continued portability to non GPU-enabled targets

Is it General?

- **Clearspeed** – two 96-wide SIMD units (MIMD/SIMD) and separate memory from host
- **Cell Blades** – four SPEs each with packed/SIMD instructions and separate memory from host, SPEs have small SW-managed local memory.
- **Larrabee** – has some number of x64 cores each with 16-wide packed/SIMD operations and separate memory from x64 host, also supporting multithreading in each core
- **Convey** – has an x64 host with an attached multi-vector accelerator which could be programmed using this model as well.
- **Multicore x64** – has some number of x64 cores, each with 4-wide packed/SIMD operations; no need for data movement?

…YES!

How does this Model apply to X64 Multicore

- Most data transfer operations and data movement and locality clauses turn into hints for data locality on x64, i.e. prefetch, temporal load and store code generator decisions
- Most directives for tuning kernel mapping turn into strip mining or tiling hints to the compiler
- Since the model is purely directive based, ignoring the directives or pragmas is a viable option

Status and Experience so Far

- Compilers have been in the field for “Technology Preview” since beginning of 2009
- Compilers generate correct x64+GPU code for most arbitrarily-chosen simple rectangular loops
- Significant speed-ups vs 1 host core on highly compute-intensive loops (matmul 20x – 30x, Black-Scholes 15x – 20x vs 1 host core)
- Compilers aggressively use NVIDIA Shared Memory
- Early limitations to offloading large loops (CUDA arg limits, live-out scalars, function calls, certain intrinsic calls – and bugs ☺) are being addressed
- Directives / clauses complete for Rel. 1.0; more to come
- Manual inlining likely to be biggest challenge for users

Limitations, Future Work

- **Not Universal** – doesn't apply to unstructured parallelism or dynamic parallelism
- **Interoperability** – between PGI Accelerator code and CUDA (e.g. CUBLAS) or OpenCL
- **Device resident data** – between kernel invocations
- **C++** – support lags C and Fortran.
- **Multiple GPUs**
- **Multiple parallel streams** – overlap computation and communication, pipelining large datasets
- **Tuning GPU-side code generation** – loop unrolling, tuning use of the SW-managed cache, tuning for single-issue in-order, etc

Conclusion: Applicability to X64

- **Feasibility: -** If the compiler supports OpenMP and Vectorization, this model is not a stretch
- **Portability:** – A key will be whether or not this model can express parallelism, tiling, and data locality in a device-independent way
- **Flexibility: -** Our model is still under active development. See http://www.pgroup.com/accelerate and give us your thoughts