



**Incremental Migration of C and Fortran
Applications to GPGPU using HMPP**
HPC Advisory Council China Conference 2010

Introduction

- Many applications can benefit from GPU computing
 - Linear Algebra, signal processing
 - Bio informatics, molecular dynamics
 - Magnetic resonance imaging, tomography
 - Reverse time migration, electrostatic
 - ...
- Porting legacy codes to GPU computing is a major challenge
 - Can be very expensive
 - Require to minimize porting risks
 - Should be based on future-proof approach
 - Implies application and performance programmers to cooperate
- A good methodology is paramount to reduce porting cost
 - HMPP provides an efficient solution

What is HMPP? (Hybrid Manycore Parallel Programming)

- A directive based multi-language programming environment
 - Help keeping software independent from hardware targets
 - Provide an incremental tool to exploit GPU in legacy applications
 - Avoid exit cost, can be future-proof solution
- HMPP provides
 - Code generators from C and Fortran to GPU (CUDA or OpenCL)
 - A compiler driver that handles all low level details of GPU compilers
 - A runtime to allocate and manage GPU resources
- Source to source compiler
 - CPU code does not require compiler change
 - Complement existing parallel APIs (OpenMP or MPI)

HMPP Main Design Considerations

- Focus on the main bottleneck
 - Communication between GPUs and CPUs
- **Allow incremental development**
 - Up to full access to the hardware features
- Work with other parallel APIs (e.g. OpenMP, MPI)
 - Orchestrate CPU and GPU computations
- Consider multiple languages
 - Avoid asking users to learn a new language
- Consider resource management
 - Generate robust software
- Exploit vendor tools/compiler
 - Do not replace, complement

How Does HMPP Differ from CUDA or OpenCL?

- HMPP parallel programming model is **parallel loop centric**
- CUDA and OpenCL parallel programming models are **thread centric**

```
void saxpy(int n, float alpha,
           float *x, float *y){
#pragma hmppcg parallel
for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```

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

int nblocks = (n + 255) / 256;
saxpy_cuda<<<nblocks,
256>>>(n, 2.0, x, y);
```

HMPP Codelets and Regions

- A codelet is a pure function that can be remotely executed on a GPU
- Regions are a short cut for writing codelets

```
#pragma hmpp myfunc codelet, ...
void saxpy(int n, float alpha, float x[n], float
y[n]){
  #pragma hmppcg parallel
    for(int i = 0; i<n; ++i)
      y[i] = alpha*x[i] + y[i];
}
```

```
#pragma hmpp myreg region, ...
{
  for(int i = 0; i<n; ++i)
    y[i] = alpha*x[i] + y[i];
}
```

HMPP Codelets Arguments

- The arguments of codelet are also allocated in the GPU device memory
 - Must exist on both sides to allow backup execution
 - No hardware mechanism to ensure consistencies
 - Size must be known to perform the data transfers
- Are defined by the io clause (in Fortran use intent instead)
 - **in** (default) : read only in the codelet
 - **out**: completely defined, no read before a write
 - **inout**: read and written
- Using inappropriate **inout** generates extra PCI bus traffic

```
#pragma hmpp myLabel codelet, args[B].io=out, args[C].io=inout
void myFunc( int n, int A[n], int B[n], int C[n]){
    for( int i=0 ; i<n ; ++i){
        B[i] = A[i] * A[i];
        C[i] = C[i] * A[i];
    }
}
```

Codelet Target Clause

- Target clause specifies what GPU code to generate
 - *GPU* can be *CUDA* or *OpenCL*
- Choice of the implementation at runtime can be different!
 - The runtime select among the available hardware and code

```
#pragma hmpp myLabel codelet, target=[GPU], args[C].io=out
void myFunc( int n, int A[n], int B[n], int C[n]){
    ...
}
```

```
#pragma hmpp myLabel codelet, target=CUDA
```

NVIDIA only GPU

```
#pragma hmpp myLabel codelet, target=OpenCL
```

NVIDIA & AMD GPU, AMD CPU

Running a Codelet or Section on a GPU - 1

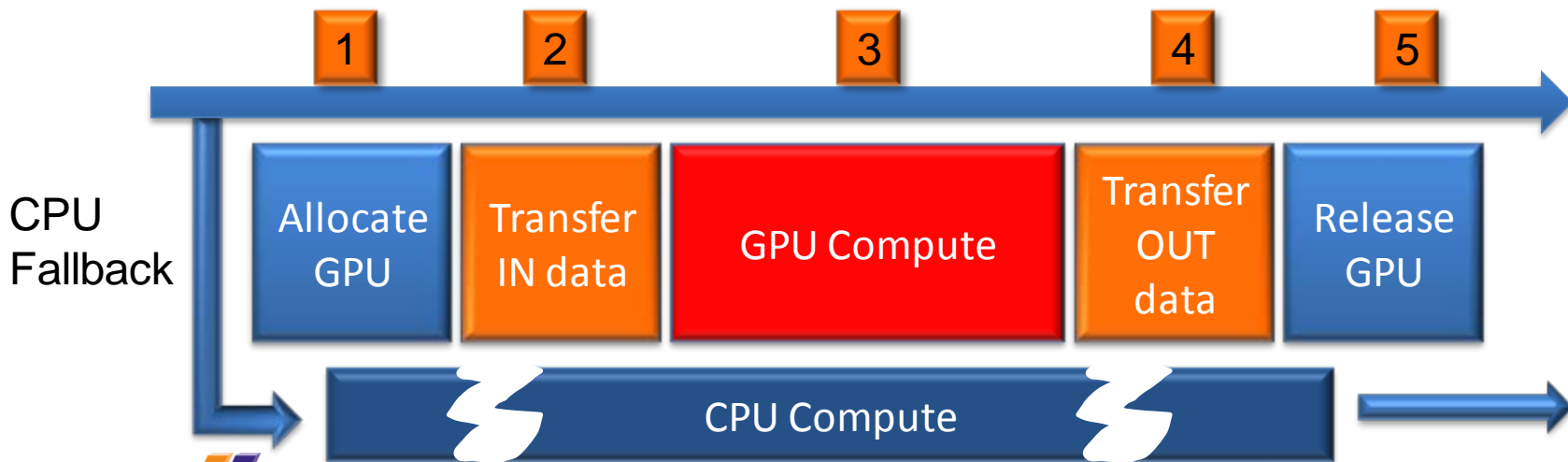
- The `callsite` directive specifies the use of a codelet at a given point in your application.
- `callsite` directive performs a Remote Procedure Call onto the GPU

```
#pragma hmpp call1 codelet, target=CUDA
#pragma hmpp call2 codelet, target=OpenCL
void myFunc(int n, int A[n], int B[n]){
    int i;
    for (i=0 ; i<n ; ++i)
        B[i] = A[i] + 1;
}

void main(void)
{
    int X[10000], Y[10000], Z[10000];
    ...
    #pragma hmpp call1 callsite, ...
    myFunc(10000, X, Y);
    ...
    #pragma hmpp call2 callsite, ...
    myFunc(1000, Y, Z);
    ...
}
```

Running a Codelet or Section on a GPU - 2

- By default, a CALLSITE directive implements the whole Remote Procedure Call (RPC) sequence
- An RPC sequence consists in 5 steps:
 - (1) Allocate the GPU and the memory
 - (2) Transfer the input data: CPU => GPU
 - (3) Compute
 - (4) Transfer the output data: GPU=> CPU
 - (5) Release the GPU and the memory



Tuning Hybrid Codes

- Tuning hybrid code consists in
 - Reducing penalty when allocating and releasing GPUs
 - Reducing data transfer time
 - Optimizing performance of the GPU kernels
 - Using CPU cores in parallel with the GPU
- HMPP provides a set of directives to address these optimizations
- The objective is to get efficient CPU **and** GPU computations

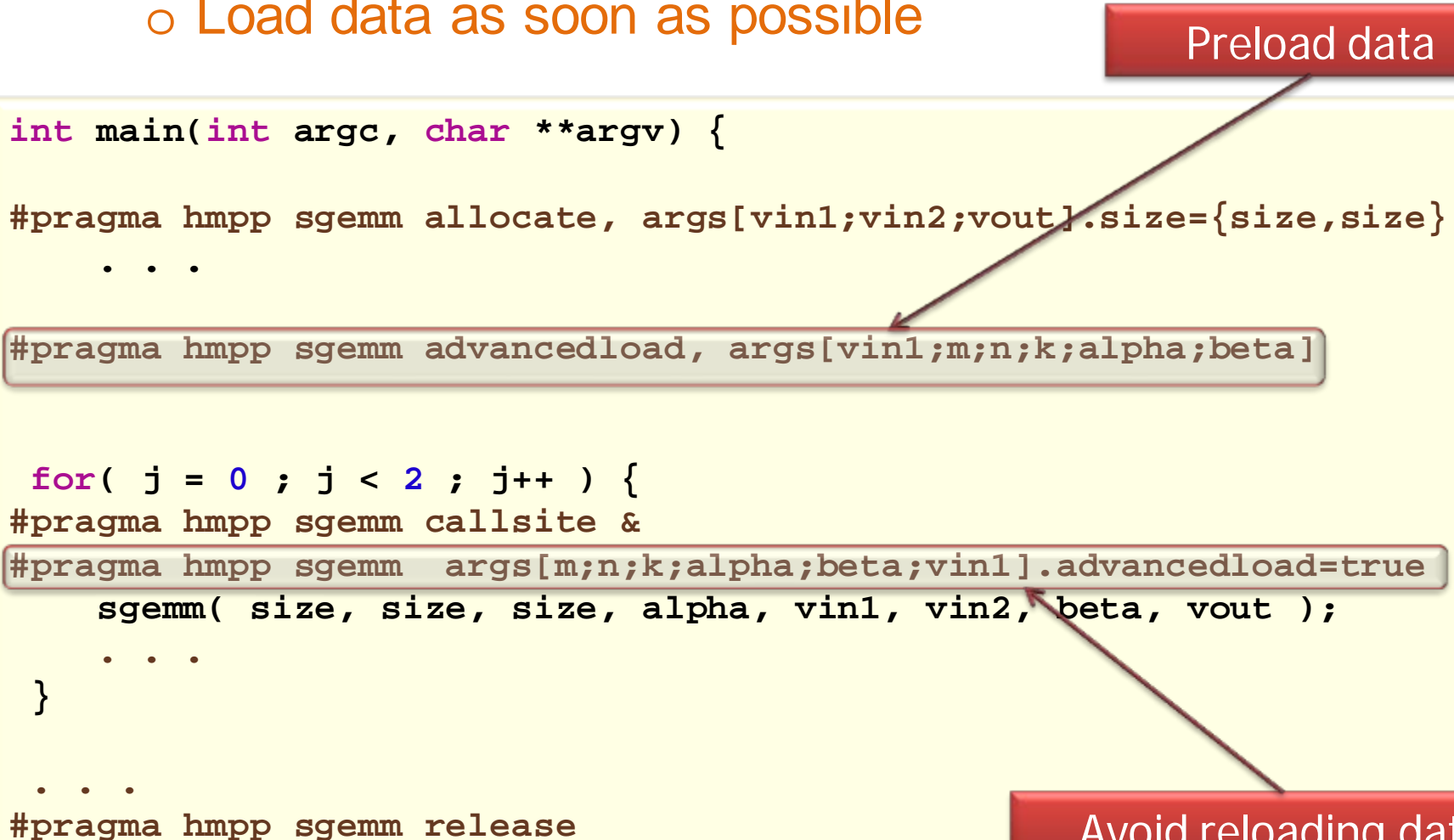
Reducing Data Transfers between CPUs and GPUs

- Hybrid code performance is very sensitive to the amount of CPU-GPU data transfers
 - PCIx bus is a serious bottleneck (< 10 GBs vs 150 GBs)
- Various techniques
 - Reduce data transfer occurrences
 - Share data on the GPU between codelets
 - Map codelet arguments to the same GPU space
 - Perform partial data transfers
- Warning: dealing with two address spaces may introduce inconsistencies

Reducing Data Transfers Occurrences

- Preload data before codelet call
 - Load data as soon as possible

```
int main(int argc, char **argv) {  
#pragma hmpp sgemm allocate, args[vin1;vin2;vout].size={size,size}  
  . . .  
#pragma hmpp sgemm advancedload, args[vin1;m;n;k;alpha;beta]  
  
  for( j = 0 ; j < 2 ; j++ ) {  
#pragma hmpp sgemm callsite &  
#pragma hmpp sgemm args[m;n;k;alpha;beta;vin1].advancedload=true  
  sgemm( size, size, size, alpha, vin1, vin2, beta, vout );  
  . . .  
  }  
  
  . . .  
#pragma hmpp sgemm release
```



Sharing Data Between Codelets with Resident Data

- Share data between codelets of the same group
 - Keep data on the HWA between two codelet calls
 - Avoid useless data transfers

```
#pragma hmpp <process> group, target=CUDA
#pragma hmpp <process> resident
float initValue = 1.5f, offset[9];
...
#pragma hmpp <process> reset1 codelet, args[t].io=out
void reset(float t[M][N]){
    int i,j;
    for (i = 0; i < M; i += 1) {
        for (j = 0; j < N; j += 1) {
            t[i][j] = initValue + offset[(i+j)%9];
        }
    }
}
#pragma hmpp <process> process codelet, args[a].io=inout
void process(real a[M][N], real b[M][N]){
    int i,j;
    for (i = 0; i < M; i += 1) {
        for (j = 0; j < N; j += 1) {
            a[i][j] = cos(a[i][j]) + cos(b[i][j]) - initValue;
        }
    }
}
```

Tuning GPU Kernels

- GPU kernel tuning set-up parallel loop suitable for GPU architectures
- Multiple issues to address
 - Memory accesses
 - Thread grid tuning
 - Register usage tuning
 - Shared memory usage
 - Removing control flow divergence
- In many cases, CPU code structure conflicts with GPU efficient code structure

Methodology to Port Applications

- Prerequisite
 - Understand your performance goal
 - Memory bandwidth needs are a good potential performance indicator
 - Know your hotspots
 - Beware of Amdahl's law
 - Ensure you know how to validate the output of your application
 - Rounding may differ on GPUs
 - Determine if your goal can be achieved
 - How many CPUs and GPUs are necessary?
 - Is there similar existing code for GPUs (in CUDA, OpenCL or HIPP)?
- Define an incremental approach
 - Ensure to check the results at each step
- Two phase approach
 - Phase 1: Application programmers validate the computed results
 - Phase 2: Performance programmers focus on GPU code tuning and data transfer reduction

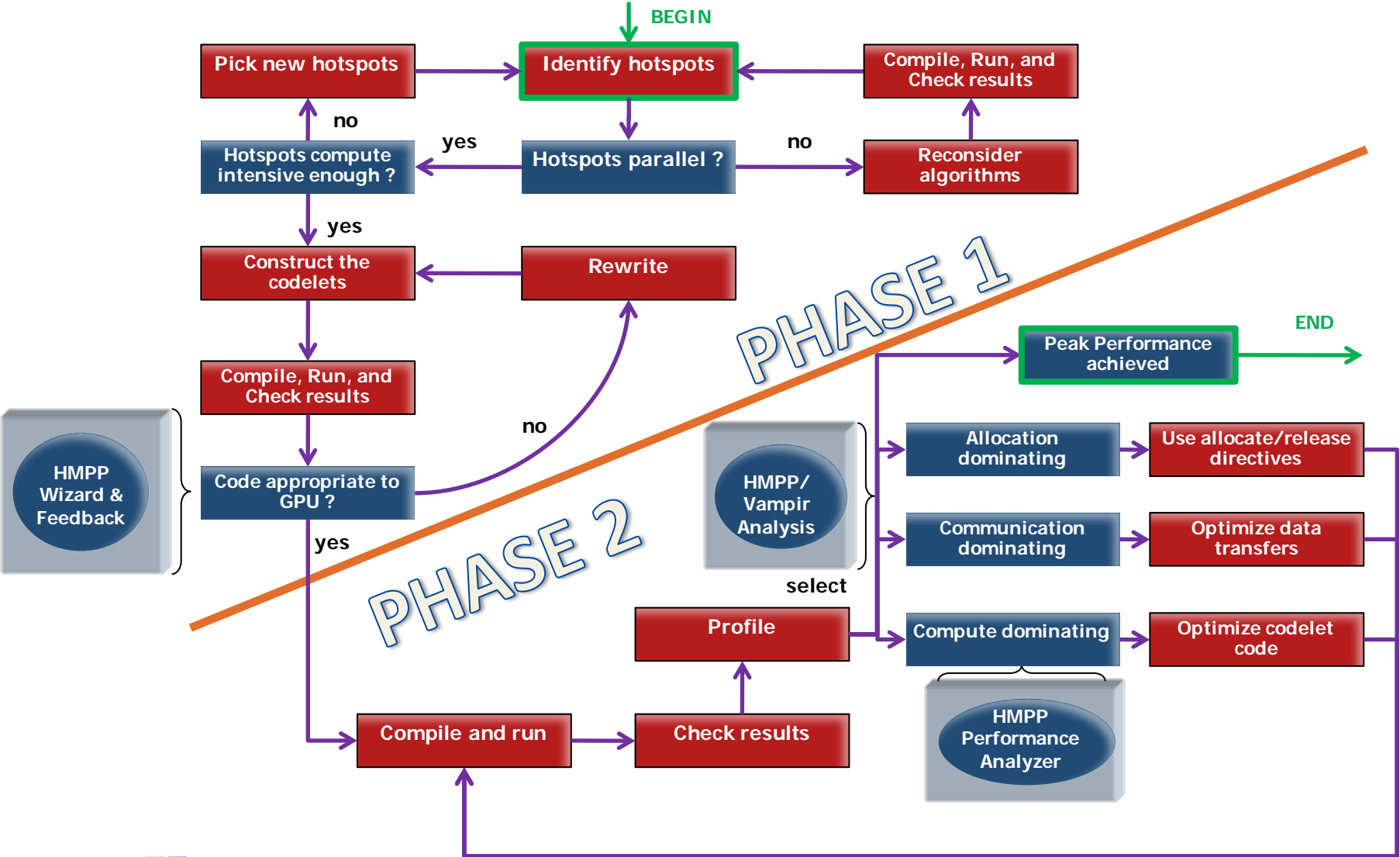
Phase 1

- Phase 1 – Dealing with application issues
 - Exhibit application SIMT parallelism
 - Push application hotspot on GPU
 - Validate execution
 - Optimize CPU code
 - Runaway from flat profile
- Phase 1 - Tools
 - CPU code profiler
 - CPU code optimizer: Vtune, Acumen, ...
 - HMPP, ...

Phase 2

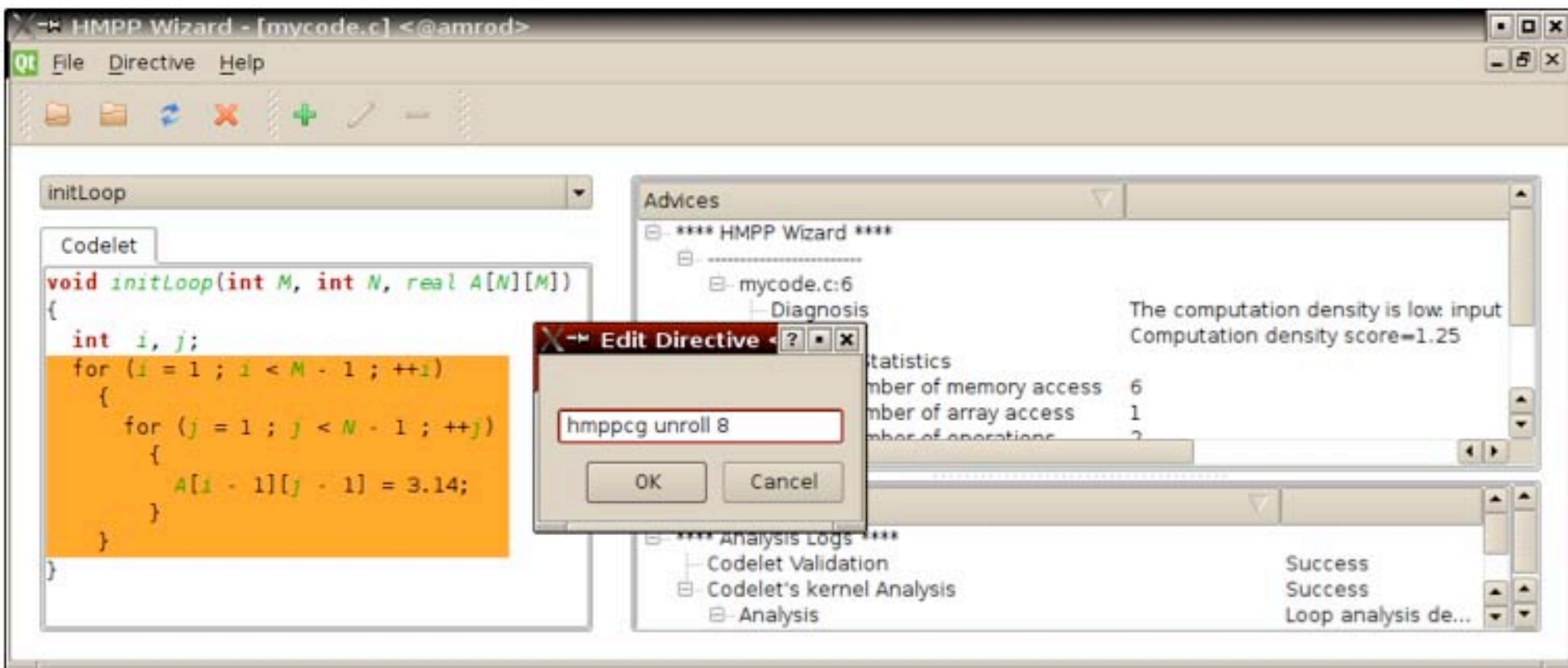
- Phase 2 – Performance
 - Optimize application GPU execution
 - Provide feedback to application programmers for improving application algorithm/data structures/...
 - Exploit CPU and GPU
 - Reduce CPU-GPU data transfers
- Phase 2 - Tools
 - GPU profiling: Vampir, Paraver, Nsight, TAU
 - Debugger: Allinea DDT, ...
 - HMPP, ...

Methodology Overview



CAPS HMPP Wizard (Beta)

- Provide advice to programmers to help them to write GPU friendly code



HMPP Performance Analyzer (Beta)

The screenshot displays the HMPP PerfAnalyzer interface for a CUDA kernel named 'dgemm'. The interface is divided into three main sections:

- Codelet:** Shows the source code for the 'dgemm' kernel. The code is annotated with HMPP pragmas and includes a loop structure for calculating matrix products. A vertical bar on the left indicates the kernel's execution progress, with 'Kernel #1 99%' highlighted in red and 'Kernel #2 0%' in green.
- Advices:** Provides performance metrics for two kernel instances. The first instance (Kernel number 1) has an average GPU execution time of 80259 us and a global memory read throughput of 8.073 GB/s/TPC. The second instance (Kernel number 2) has an average GPU execution time of 208.773 us and a global memory read throughput of 2.645 GB/s/TPC. Detailed metrics for each kernel are also listed.
- Analysis Log:** Shows the results of the analysis, including 'Codelet Validation' (Success), 'Profile Analysis' (Success), and 'Profile Files' (cuda_profile_0.csv, cuda_profile_1.csv, cuda_profile_2.csv).

Examples of Ported Applications – 1

- Smoothed particles hydrodynamics

- Effort: 2 man-month
- Size: 22kLoC of F90 (SP or DP)
- GPU C1060 improvement: x 2 over 1 Nehalem core (x1.1 DP)
- Main difficulty: kernels limited to 70% of the execution time

The ratio performance over resource is the important information here.

- 3D Poisson equation, conjugate gradient

- Effort: 2 man-month
- Size: 2kLoC of F90 (DP)
- CPU improvement: x 2
- GPU C1060 improvement: x 5 over 1 Nehalem core
- Main porting operation: highly optimizing kernels
- Main difficulty: none

Examples of Ported Applications - 2

- Electron propagation - solver
 - Effort: 2 man-month
 - Size: 10 kLoC of F95 (DP, MPI)
 - CPU improvement: x 1.3
 - GPU C1060 improvement: x 1.15 over 4 Nehalem cores
 - Main porting operation: solver algorithm modifications
 - Main difficulty: small matrices, many data transfers
- 3D combustion code
 - Effort: 2 man-month
 - Size: ~1MLoC of F90 (DP)
 - GPU C1060 improvement: ~x1 (data transfer limited) over 1 Nehalem core; C2050 x1.3
 - Main difficulty: execution profile shows few hot-spots (70%)
 - Next: hybrid parallelization

Examples of Ported Applications - 3

- Euler equations
 - Effort: <1 man-month
 - Size: ~40kLoC of F90 (DP)
 - CPU improvement: x 3 over the original code
 - GPU C1060 improvement: x 3 over 1 Nehalem core
 - Main porting operation: specializing the code for the main execution configuration
 - Main difficulty: reorganizing computational kernels
- Tsunami/flood simulation
 - Effort: 0.5 man-month
 - Size: ~4kLoC (DP, MPI)
 - GPU C1060 improvement: x 1.28 over 1 Nehalem core
 - Next: highlight more parallelism, reducing data transfers (High performance potential)

Examples of Ported Applications - 4

- Molecular fluid dynamics
 - Effort: 1 man-month
 - Size: ~1kLoC of C99 (DP)
 - GPU C2050 improvement: ~x 2 over 4 Nehalem cores
 - Main porting operation: pointer based data structure rewriting
 - Main difficulty: Very efficient original OpenMP code
- Monte Carlo simulation for thermal radiation
 - Effort: 1.5 man-month
 - Size: ~1kLoC of C d (DP)
 - GPU C2050 improvement: x6 over 8 Nehalem cores
 - Full hybrid version: x23 with 8 nodes over 8 Nehalem cores
 - Main porting operation: adding a few HMPP directives
 - Main difficulty: none

Examples of Ported Applications - 5

- Weather models (GTC 2010 talk, M. Govett, NOAA)
 - Effort: 1 man-month (part of the code already ported)
 - GPU C1060 improvement: 10 over a Nehalem CPU
 - Main porting operation: reduction of CPU-GPU transfers
 - Main difficulty: GPU memory size is the limiting factor

Conclusion

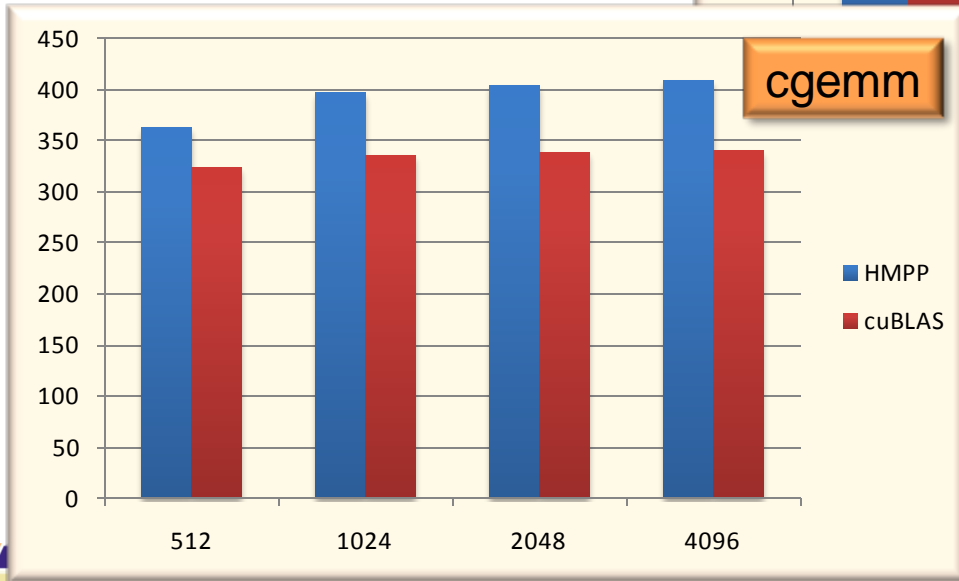
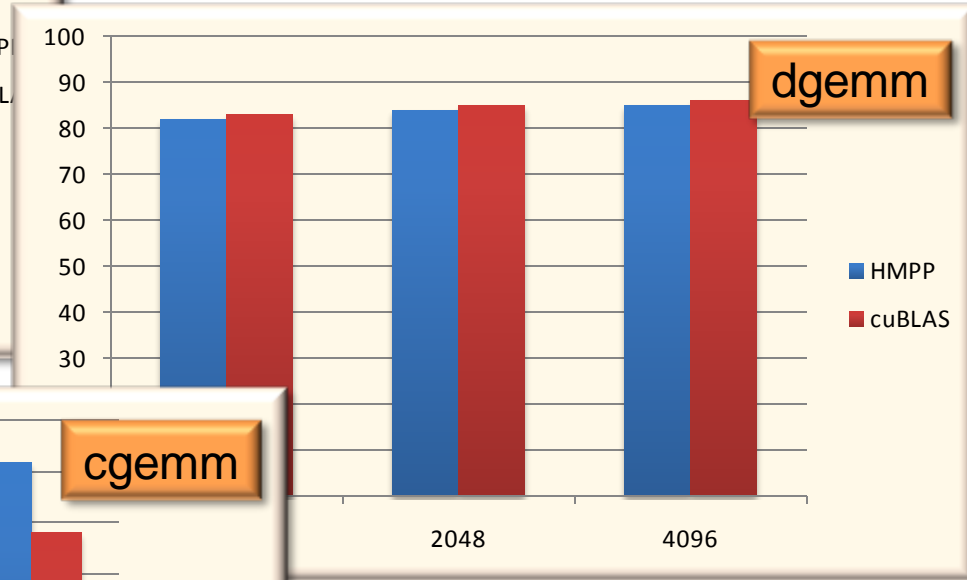
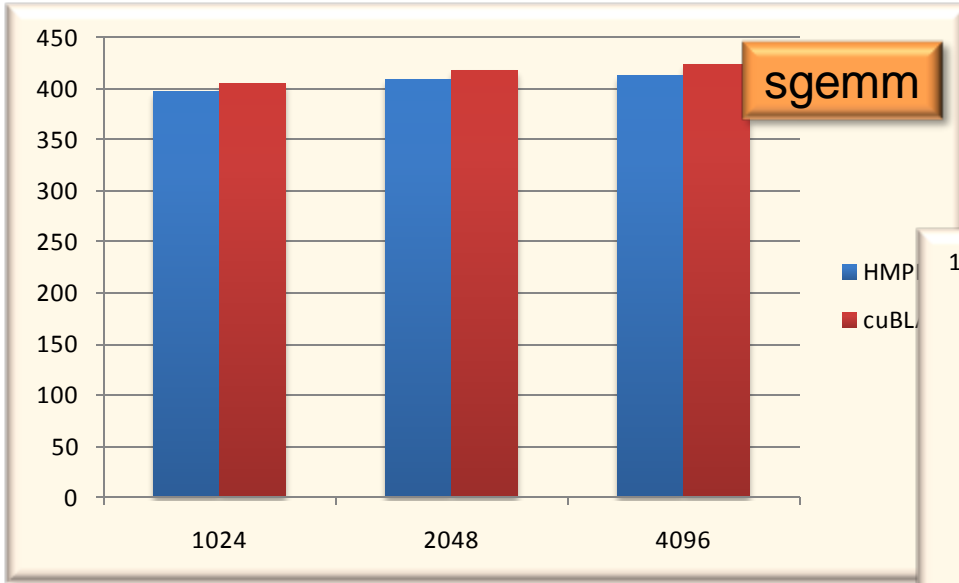
- Heterogeneous architectures are becoming ubiquitous
 - In HPC centers but not only
 - Tremendous opportunities but not always easy to seize
 - CPU and GPU have to be used simultaneously
- Legacy codes still need to be ported
 - An efficient methodology is required
 - A methodology supporting tools is needed and must provide a set of consistent views
 - The legacy style is not helping
 - Highlighted parallelism for GPU is useful for future manycores
- HMPP based programming
 - Helps implementing incremental strategies
 - Is being complemented by a set of tools
 - Engage in an Open Standard path with Pathscale



CAPS'''

Innovative Software for Manycore Paradigms

S/D/CGEMM Performance Example (NVIDIA GTX 275)



gigaflops x matrix size