HMPP

A directive-based compiler for GPU-accelerated applications

Stéphane BIHAN

HealthGrid Conference 2010

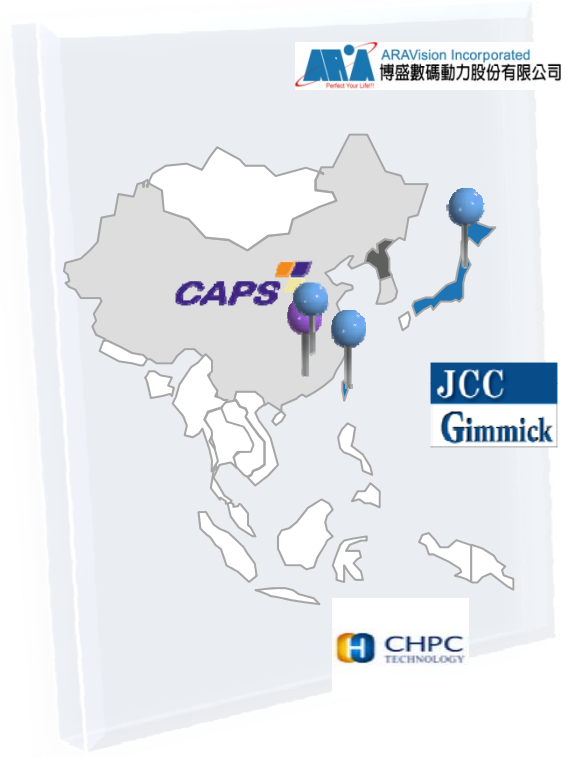
CAPS

CAPS Profile

CAPS Background

- Created in 2002
  - Strong expertise in processor micro-architecture and code generation
  - Spin-off French INRIA Research Lab
- Mission: to help its customers to leverage the computing power of multi/manycores
  - HMPP™ software editor
  - Consulting and engineering services
  - Trainings
- 30 employees

CAPS Worldwide



- Offices
  - Headquarters: Rennes, France
  - Shanghai, China
- Resellers and Partners
  - US: ParaTools, SGI
  - EU: C-S, SGI
  - APAC: JCC-Gimmick (Japan), Aravision (Taiwan), CHPC (China)

References


AWE


cea


HLRS


National Taiwan University
Civil Engineering


EADS


ifp


長庚技術學院
CHANG GUNG INSTITUTE OF TECHNOLOGY


國立交通大學
National Chiao Tung University


Ifremer


OAK RIDGE
National Laboratory


TOTAL


東北大学
TOHOKU UNIVERSITY


KEK


ONERA
THE FRENCH AEROSPACE LAB


GENCI


東京工業大学
Tokyo Institute of Technology

HMPP

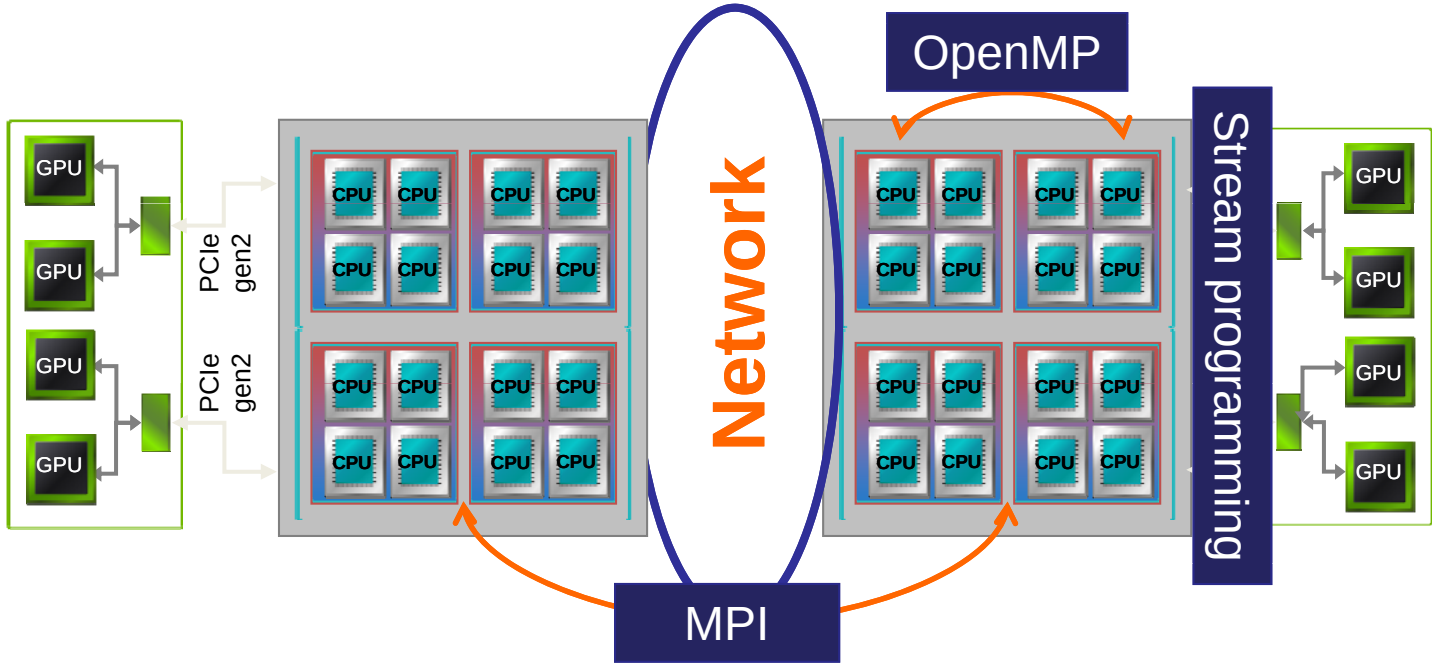
Overview

Multi/Many Cores

- Homogeneous cores
  - Intel Xeon 8 cores, QPI
  - AMD Magny-Cours up to 12 cores, HT
  - About 100 Gflops DP
- GPU compute processors
  - NVIDIA Fermi: ~1.5 Tflops SP, ~800 Gflops DP
  - AMD FireStream: 2.6 Tflops SP, 500 Tflops DP (Radeon 5870)
  - Toward manycores: Intel Many Integrated Core, AMD Fusion
- Programming models and tools
  - NVIDIA CUDA, OpenCL, Intel Ct
  - Directive-based compilers: HMPP, PGI, PathScale

Multiple Levels of Parallelism

- Amdahl's law is forever, all levels of parallelism need to be exploited
- Programming various hardware components of a node cannot be done separately



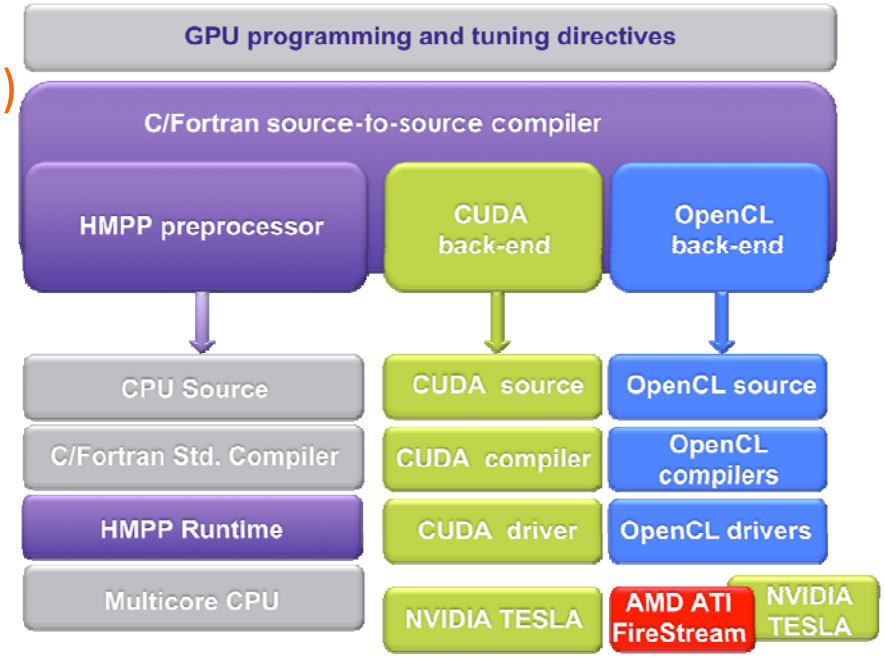
HMPP Objectives

To give developers
a high level abstraction for
manycore programming

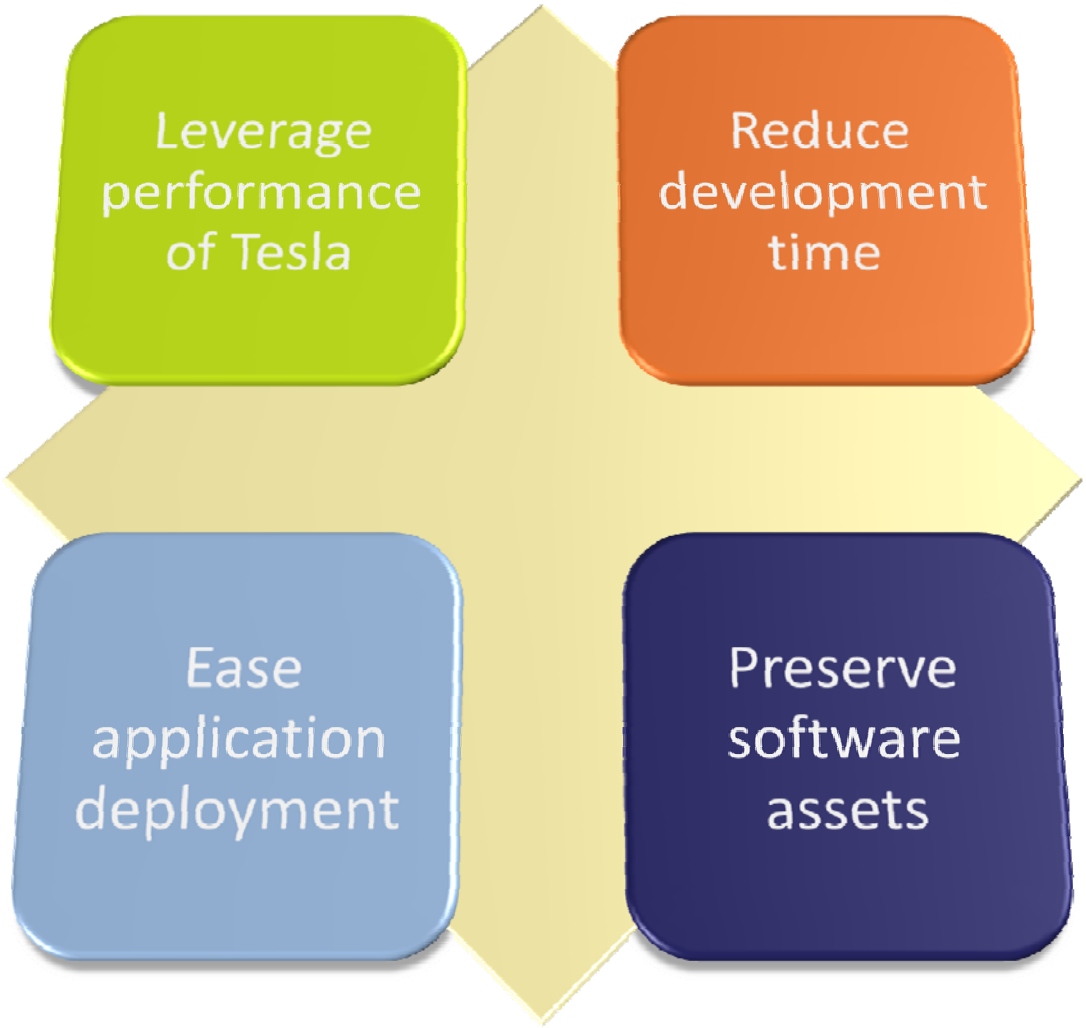
- Incrementally program GPU-accelerated applications
- Rapidly build hybrid applications
- Keep accelerated kernels hardware independent
- Ensure application portability and interoperability

Overview

- C and Fortran GPU programming directives
  - Define and execute GPU-accelerated versions of code
  - Optimize CPU-GPU data movement
  - Complementary to OpenMP and MPI
- A source-to-source hybrid compiler
  - Generate powerful accelerated kernels
  - Works with native compilers and target tools (Intel, GNU, PGI, …)
  - Tuning directives to optimize accelerated kernels
- A runtime library
  - Dispatch computations on available GPUs
  - Scale to multi-GPUs systems



HMPP Key Benefits



HMPP

Programming Directives

Accelerate Codelet Function

- Declare a GPU-accelerated version of a function

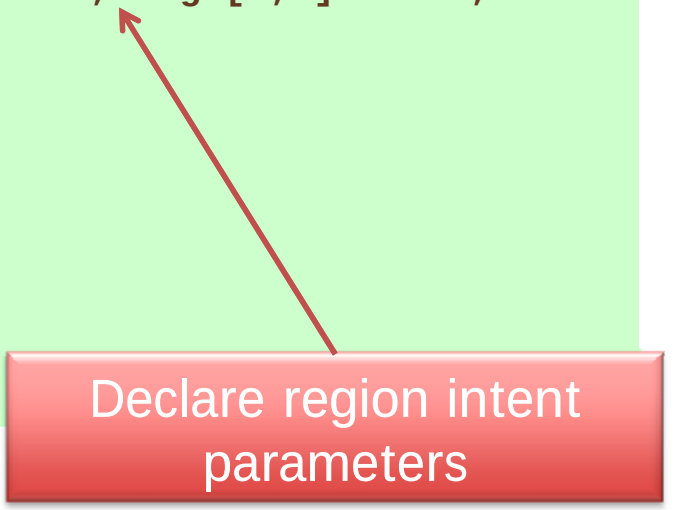
```
#pragma hmpp sgemm codelet, target=CUDA:OPENCL, args[vout].io=inout
extern void sgemm( int m, int n, int k, float alpha,
                   const float vin1[n][n], const float vin2[n][n],
                   float beta, float vout[n][n] );


int main(int argc, char **argv) {
    /* . . . */

 for( j = 0 ; j < 2 ; j++ ) {
#pragma hmpp sgemm callsite
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
  }
    /* . . . */
 }
```

Declare CUDA and OPENCL codelets

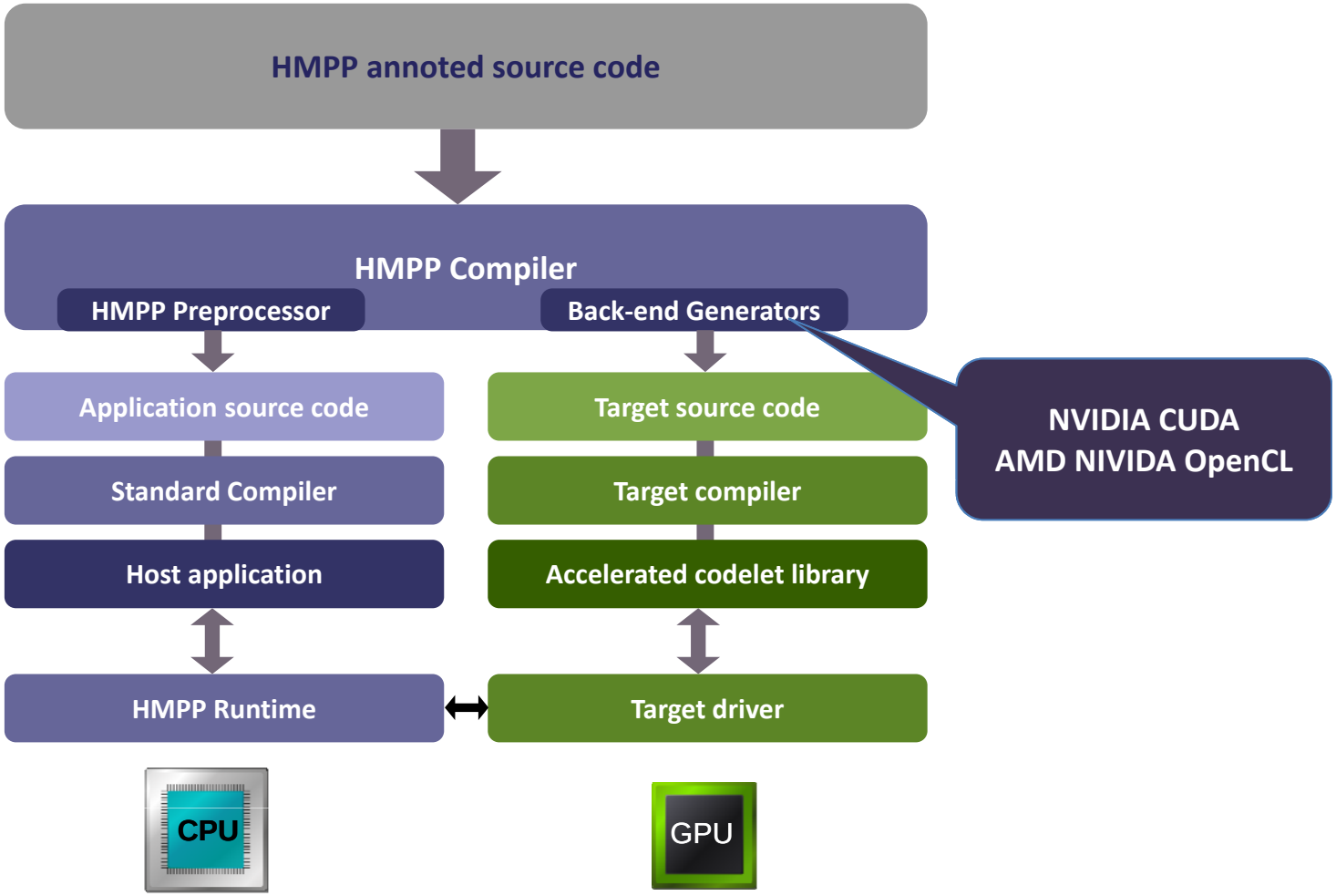
Synchronous codelet call

Accelerate Regions

```
!$hmpp MyRegion region, target=CUDA, args[A].io=inout, args[B;C].io=in,
private=[i,j]

 DO i=1,N
    DO j=1,N
       A(i,j) = A(i,j) + B(i,j) * C(i,j) ;
    ENDDO
 ENDDO

!$hmpp MyRegion endregion
```



Compilation Workflow



Allocate and Release

- Avoid initializing and allocating device each codelet call

```
int main(int argc, char **argv) {
    /* . . . */
#pragma hmpp sgemm allocate, args[vin1;vin2;vout].size={size,size}
    /* . . . */

 for( j = 0 ; j < 2 ; j++ ) {
#pragma hmpp sgemm callsite
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    /* . . . */
  }

    /* . . . */
#pragma hmpp sgemm release
}
```

Allocate and initialize device early

Release device

Optimize Data Movement

- Preload data before codelet call

```
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
```

Preload data

Avoid reloading data

Compute Asynchronously

- Perform CPU/GPU computations asynchronously

```
int main(int argc, char **argv) {
    /* . . . */
#pragma hmpp sgemm allocate, args[vin1;vin2;vout].size={size,size}
    /* . . . */

 for( j = 0 ; j < 2 ; j++ ) {
#pragma hmpp sgemm callsite, asynchronous
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    /* . . . */
  }

    /* . . . */
#pragma hmpp sgemm synchronize
#pragma hmpp sgemm delegatedstore, args[vout]
#pragma hmpp sgemm release
}
```

Execute asynchronously

Download result when needed

Group of Codelets

- Optimize CPU-GPU data movement
- Codelets can share variables
  - Keep data in GPUs between two codelets
  - Avoid useless data transfers
  - Map arguments of different functions in same GPU memory location (equivalence Fortran declaration)

More Flexibility and Performance

Data Mapping Example

- Share data between codelets of same group

```
#pragma hmpp <mygp> group, target=CUDA
#pragma hmpp <mygp> map,   args[f1::inm; f2::inm]

#pragma hmpp <mygp> f1 codelet, args[outv].io=inout
static void matvec1(int sn, int sm,
                    float inv[sn], float inm[sn][sm], float outv[sm])
{
  ...
}
#pragma hmpp <mygp> f2 codelet, args[v2].io=inout
static void otherfunc2(int sn, int sm,
                       float v2[sn], float inm[sn][sm])
{
  ...
}
```

Data share the same memory space on the device

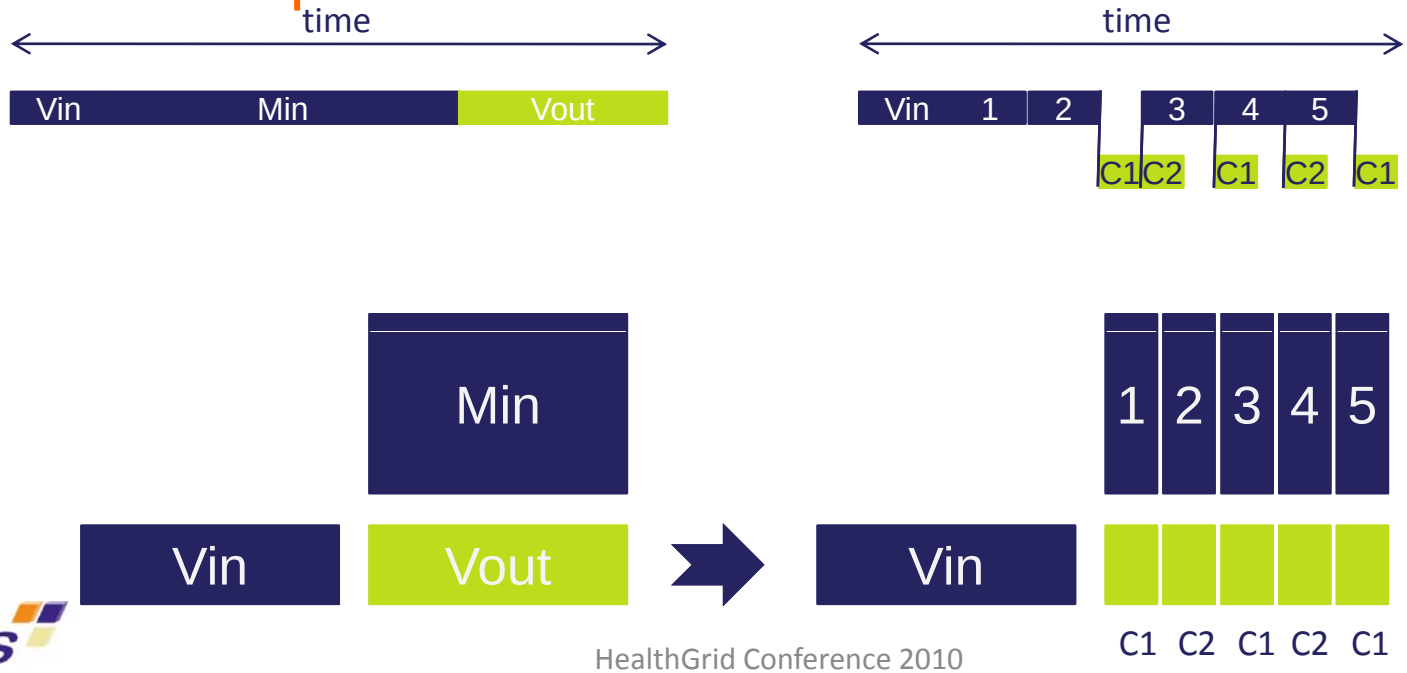
Overlapping Example

Codelet1 execution overlaps codelet2 data transfers

```
#pragma hmpp <sgemmDB> c1 callsite, asynchronous
  sgemm(m,n/2,k, alpha, vin1, vin2, beta, vout );

#pragma hmpp <sgemmDB> c2 advancedload, args[vin1;vout], asynchronous

#pragma hmpp <sgemmDB> c1 synchronize
#pragma hmpp <sgemmDB> c1 delegatedstore, args[vout]

#pragma hmpp <sgemmDB> c2 callsite
  sgemm(m,n/2,k,alpha,vin1,&vin2[n/2*k],beta,&vout[n/2*m]);
```

Double Buffering

- Pipeline GPU kernel execution with data transfers
  - Split single function call in two codelets (C1, C2)
  - Overlap with data transfers



HMPP

Tuning Directives

Code Generation Optimizations

- Add code properties
  - Force loop parallelization
  - Indicate parameter aliasing
- Apply code transformations
  - Loop unrolling, blocking, tiling, permute, …
- Control mapping of computations
  - Gridification
  - Use of GPU shared memory
  - GPU threads synchronization barriers

Tuning Directive Example

```
#pragma hmpp dgemm codelet, target=CUDA, args[C].io=inout
void dgemm( int n, double alpha, const double *A, const double *B,
            double beta, double *C ) {
int i;

#pragma hmppcg(CUDA) grid blocksize "64x1 »
#pragma hmppcg(CUDA) permute j,i
#pragma hmppcg(CUDA) unroll(8), jam, split, noremainder
#pragma hmppcg parallel
 for( i = 0 ; i < n; i++ ) {
   int j;
#pragma hmppcg(CUDA) unroll(4), jam(i), noremainder
#pragma hmppcg parallel
   for( j = 0 ; j < n; j++ ) {
     int k; double prod = 0.0f;
     for( k = 0 ; k < n; k++ ) {
       prod += VA(k,i) * VB(j,k);
     }
     VC(j,i) = alpha * prod + beta * VC(j,i);
   }
 }
}
```

1D gridification
Using 64 threads

Loop transformations

Using Shared Memory

```
#define N (256*10+2*DIST)
void conv1(int A[N], int B[N])
{
 int i,k ;
 int buf[DIST+256+DIST] ;
 int grid = 0 ;
#pragma hmppcg set grid = GridSupport()
If grid){
#pragma hmppcg grid blocksize 256x1
#pragma hmppcg parallel
   for (i=DIST; i<N-DIST ; i++){
#pragma hmppcg grid shared buf
     int t ;
#pragma hmppcg set t = RankInBlock(i)

// Load the first 256 elements
   buf[t] = A[i-DIST] ;
// Load the remaining elements
     if (t < 2*DIST )
       buf[t+256] = A[i-DIST+256] ;

#pragma hmppcg grid barrier
```

Set buffer size according to grid size

Detect grid support

Declare buf in shared memory

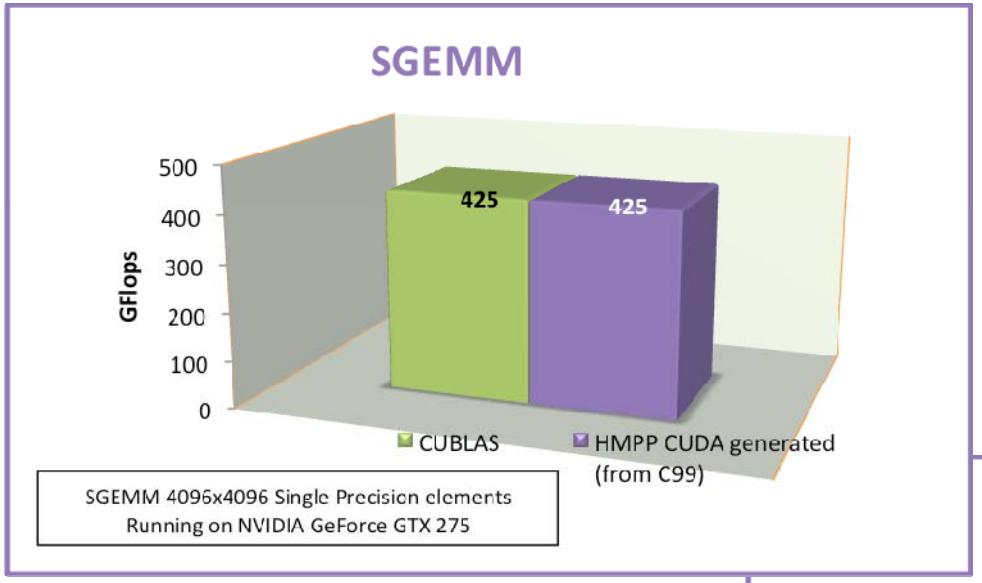
Parallel load in shared memory

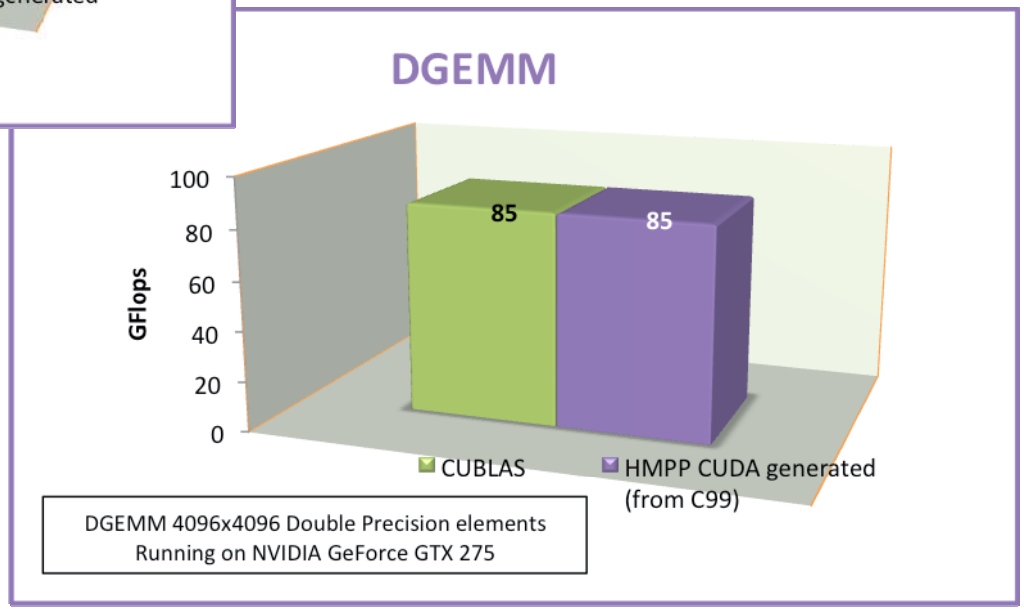
Wait for end loading before use

HMPP

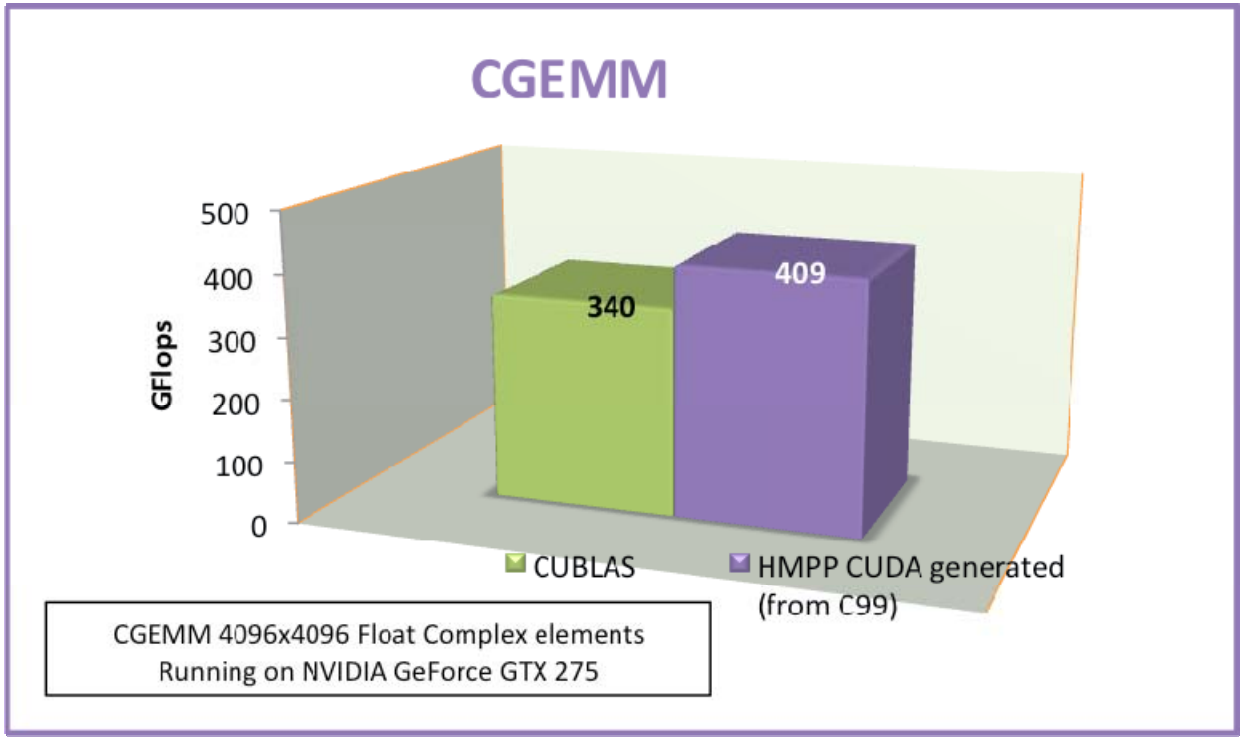
Performance Figures

HMPP CUDA – SGEMM & DGEMM





HMPP CUDA – CGEMM



Conclusion

- Abstract the programming of GPUs
  - A rich set of directives for programming and tuning GPU-accelerated applications
  - Offer incremental levels of programming from minimal to advanced and expert
- A source-to-source C and Fortran compiler
  - Work with standard compilers and hardware vendors tools
  - Extract and insulate hardware-specific computations
  - Rapidly build and tune CUDA- and OpenCL-accelerated applications
  - Preserve legacy code
- CAPS to promote HMPP as an open standard
  - PathScale has adopted the HMPP directives in its ENZO compiler

Innovative Software for Manycore Paradigms