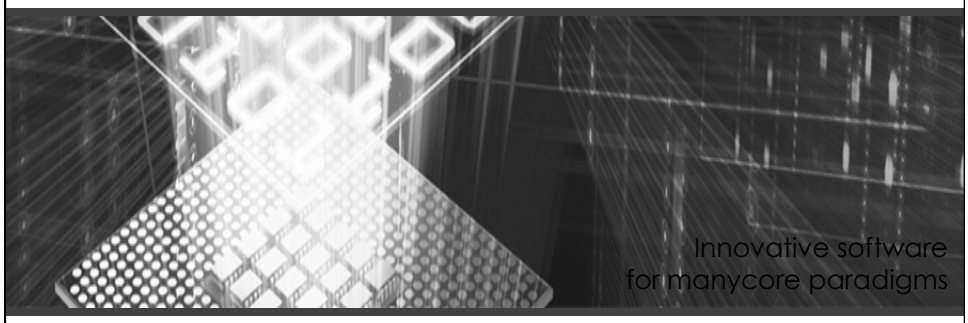
CAPS



hmpp

Heterogeneous Multicore Parallel Programming

S. Chauveau & L. Morin & F. Bodin

Introduction

- Numerous legacy applications can benefit from GPU computing
- Many programming solutions are needed to allow incremental and efficient code development
- High level, directive based, approaches target new users and aim at providing standard C / Fortran GPU programming
- Heterogeneous Multicore Parallel Programming (HMPP™) has been designed to exploit GPUs in legacy codes

Why HMPP?

- GPU computing shown to be a great opportunity for many applications
  - Performance is critical in many applications fields
- Software needed for heterogeneous targets
  - Can we make it very simple?
  - Can we address a large fraction of programmers?
- Recognize a technology transition period before standards settle
  - Parallel computing still evolving fast
- Supplement existing parallel APIs

HMPP Overview

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 API (OpenMP, MPI)
  - Do not oppose GPU to CPU,
- Consider multiple languages
  - Avoid asking users to learn a new language
- Consider resource management
  - Generate robust software
- Exploit HWA constructors programming tools
  - Do not replace, complement

HMPP Basis

- Directive based approach
  - Do not require a new programming language
    - And can be applied to many based languages
  - Already state of the art approach (e.g. OpenMP)
  - Keep incremental development possible
- Portability
  - To various hardware accelerators (HWAs) and host code is independent of the HWA
- Avoid exit cost

What is missing in OpenMP for HWA?

- Remote Procedure Call (RPC) on a HWA
  - Code generation for GPU, …
  - Hardware resource management
- Dealing with non shared address spaces
  - Explicit communications management to optimize the data transfers between host and the HWA

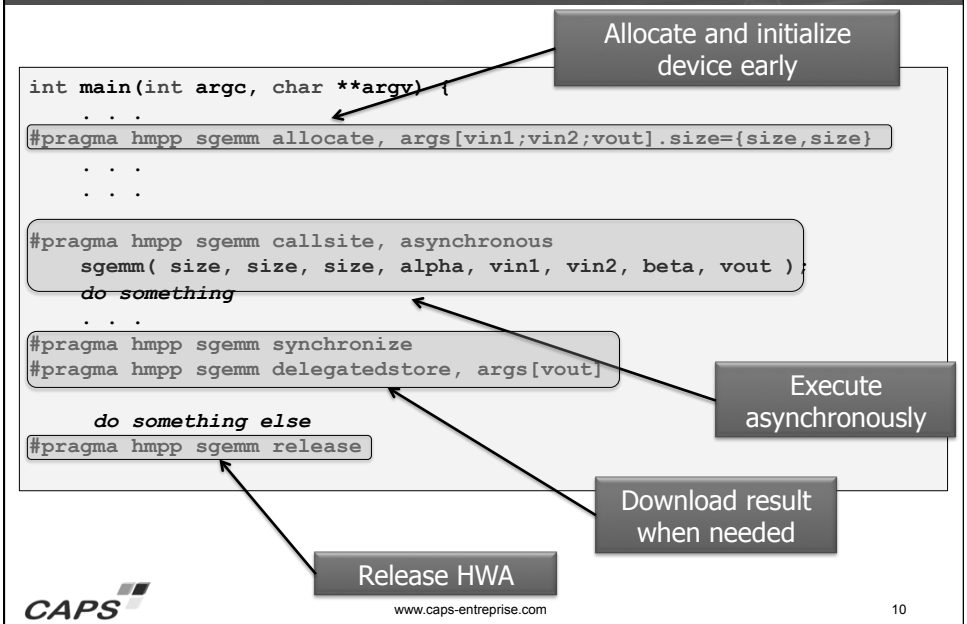
HMPP

- Bring remote procedure call on GPU
  - Host & compute device model
  - Synchronous and asynchronous
- Allow to optimize communication between CPU and GPU
- Provide high level GPU code generation from sequential code
- Provide device resource management

HMPP Example

```
#pragma hmpp sgemmlabel codelet, target=CUDA, 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 sgemmlabel callsite
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
 }
```


HMPP Example



Optimizing Communications in Groups

- Allow to exploit
  - Communication / computation overlap
  - Temporal locality of RPC parameters
- Various techniques
  - Advancedload and delegatedstore
  - Constant parameter
  - Resident data
  - Actual argument mapping
  - Partial data transfers

HMPP Example (3)

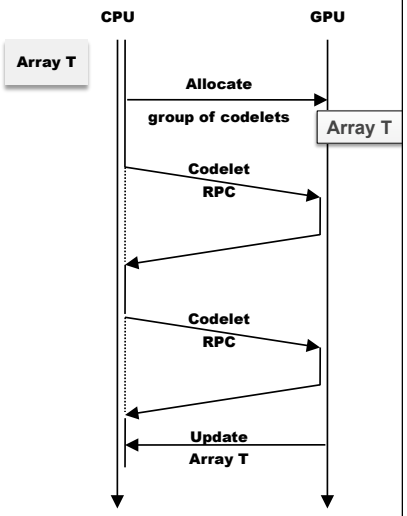
Preload data

```
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
```

Avoid reloading data

Group of Codelets (V2.0)

- Several callsites grouped in a sequence corresponding to a given device
- Memory allocated for all arguments of all codelets
- Allow for resident data without consistency management



Actual Argument Mapping

- Allocate arguments of various codelets to the same memory area
  - Allow to exploit reuses of argument to reduce communications
  - Close to equivalence in Fortran

```
#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])
{
  ...
}
```

Arguments share the same space on the HWA

HMPP Regions (V2.3)

- Reduce code restructuring when using HMPP
- The codelet is automatically built from the region code

```
#pragma hmpp region
{
  int i;
  for (i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i]*alpha;
  }
}
```


Double Buffering Example

```
…
for (index=0; index < CHUNKS; index+=2)  {
 #pragma hmpp <simple> f1 callsite, ..., asynchronous
    matvec(N, chunksize, &t3[index*chunksize], t1, &t2[N*index*chunksize]);

 #pragma hmpp <simple> f2 advancedload, … ,asynchronous

 #pragma hmpp <simple> f1 synchronize
 #pragma hmpp <simple> f1 delegatedstore, args[outv]

 #pragma hmpp <simple> f2 callsite, …, asynchronous
   matvec(N, chunksize, &t3[(index+1)*chunksize], t1, &t2[N*(index+1)*chunksize]);

   if (index+2 < CHUNKS) {
      #pragma hmpp <simple> f1 advancedload, args[outv; inm], …,asynchronous
   }

 #pragma hmpp <simple> f2 synchronize
 #pragma hmpp <simple> f2 delegatedstore, args[outv]
}
…
```

chunksize needs tuning to achieve perfect overlap

Other HMPP Features (V2.2)

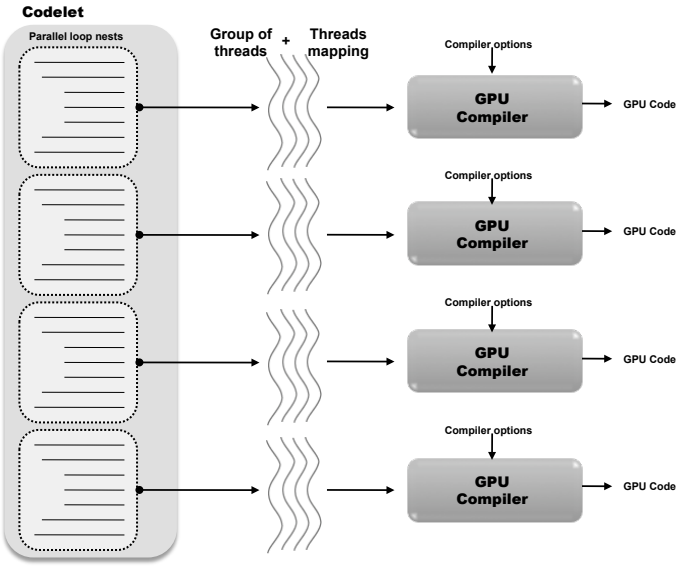
- Fallback management when GPU is busy (or unavailable)
- Provide ways to deal with machine specific features
  - e.g. pin memory, …
- Windows and Linux versions
- Seamless integration of GPU libraries such as cuFFT an cuBLAS
- GPU code tuning directives

High Level GPU Code Tuning

High Level GPU Code Tuning

- High level GPU code generation aims at removing low-level implementation details
  - But users still need to understand the underlying process
- Two different sets of tuning issues
  - Global to application: CPU-GPU data transfers
  - Local to threads: GPU kernels
- Many parameters to deal with when exploring the thread optimization space
  - Easier to explore at higher code level
  - Many useful loop transformations
  - Interaction with thread layout on the streaming multiprocessors

High Level Code Generation



Iteration Space to Thread Mapping

- Parallel iterations spaces are transformed into set of blocks of threads
- Selecting the iteration space fixes the number of threads and their mapping
- Changing to the iteration spaces impact on the threads processing
- Changes to the loop bodies modify the threads computations

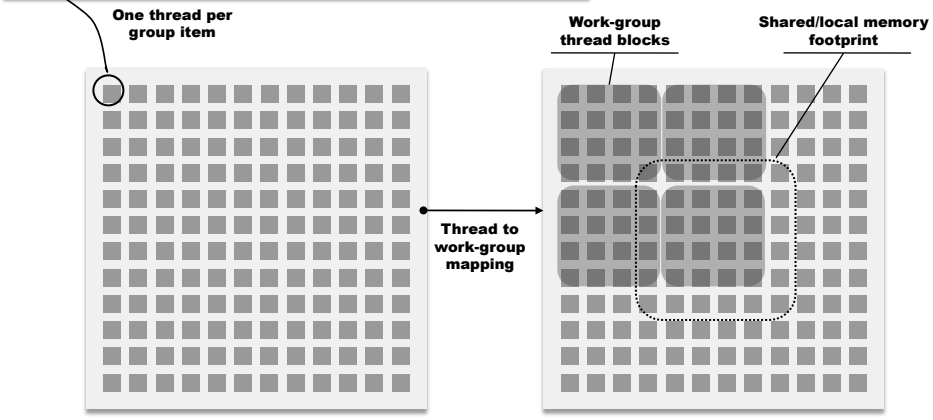
Thread body

```
!$hmppcg parallel
DO i=0,n
!$hmppcg noparallel
   DO j=1,4
      A(j,i) = . . .
   ENDDO
ENDDO
```

CAPS

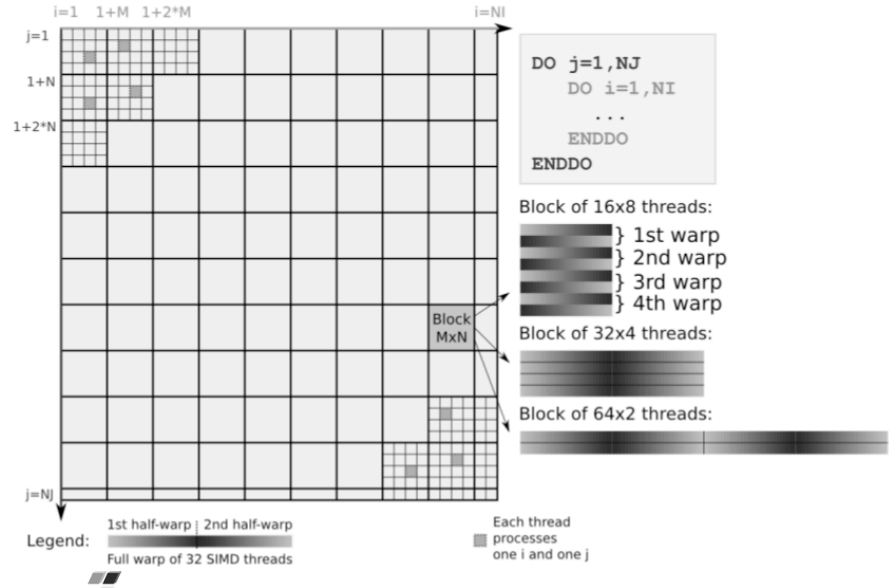
Iteration Space Mapping and Shared Memory Usage

```
for (i = 1; i < m-1; ++i){
  for (j = 1; j < n-1; ++j) {
    B[i][j] =  c11*A[i-1][j-1]+c12*A[i+0][j-1]+c13*A[i+1][j- 1]
              +c21*A[i-1][j+0]+c22*A[i+0][j+0]+c23* A[i+1][j+0]
              +c31*A[i-1][j+1]+c32*A[i+0][j+1]+c33* A[i+1][j+ 1];
  }
}
```



CAPS

Iteration Space Mapping and CUDA Threads WARP



Iteration Space Mapping and Coalesced Memory Accesses

- `A(i,k)` is well coalesced (`i` on 1st array dimension)
- `C(j,i)` is badly coalesced (`i` on 2nd array dimension)
  - But that does not really matter here
- `B(k,j)` is also well coalesced

```
DO j=1,m   ! 2nd grid dimension
 DO i=1,n  ! 1st grid dimension (the warps)
  tmp = 0
  DO k=1,n
    tmp = A(i,k)*B(k,j)
  ENDDO
  C(j,i) = tmp
 ENDDO
ENDDO
```

A Need for New Tuning Strategies

- Vector had its rules
  - Vector stride one access, …
- Superscalar SMP had its rules
  - Exploit temporal locality, …
  - Few number of threads
- Thread/stream computing need *new* rules
  - Massive number of threads
  - Explicit use of memory hierarchy
  - Data locality optimization between threads
  - …

A Set of Tuning Techniques

- Loop transformations can be used to express many of the optimizations
  - Loop permutation
  - Unroll & Jam
  - Loop Collapsing
  - …
- Use of multiprocessor shared memory
- Thread blocks size tuning

All these tuning techniques are coupled

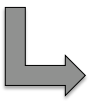
A Simple Tuning Strategy

1. Improve memory coalescing
   - Choose loop order
2. Grid size tuning
   - Choose a grid block size
3. Shared memory
   - Exploit data reuse (temporal data locality)
4. Register exploitation
   - Unroll (and Jam)

Loop Permutation

- Select the order of the loops
  - Outer loops define the iteration space
- Modify WARP dimension and coalescing properties

```
for (k = 0; k < p; k++){
      for (j = 1; j < n-1; j++) {
        //loop body
     }
 }
```



```
for (j = 1; j < n-1; j++) {
     for (k = 0; k < p; k++){
        //loop body
     }
 }
```

A Small Experiment with Loop Permutations

```
# pragma hmppcg grid blocksize %(bsx)d X %(bsy)d
# pragma hmppcg parallel
  for (i = 1; i < %(m)d-1; ++i){
    # pragma hmppcg parallel
     for (j = 1; j < %(n)d -1; ++j) {
       # pragma hmppcg parallel
        for (k = 0; k < %(p)d; ++k){
           B[i][j][k] = c11 * A[i - 1][j - 1][k]  +  c12 * A[i + 0][j -1][k]  +] . . . ;
        }
     }
  }
```

- Questions to answer
  - What is the best loop order (6 possibilities)?
  - What is the best grid configuration (24 tested here, 8x1 to 512x1)?

Example of performance for one data size

	JIK	JKI	IJK	IKJ	KJI	KIJ
Min Perf	0,8	5	0,1	10	0,1	0,8
Max Perf	1	14,4	0,2	15,9	0,5	3

CAPS

Loop Unrolling

- Reduce the cost of iterating (esp. on small loops)
- Increase the amount of work per thread
- Decrease the number of threads
- Add opportunities for local optimizations (e.g. redundant loads)
- Increase pending memory accesses to amortize long latencies of memory loads
- ...

```
#pragma hmppcg unroll(3), noremainder
for (i=0;i<90;i++) {
  // calc(i)
}
```

```
for(i=0;i<90;i+=3) {
   // ... calc(i)
   // ... calc(i+1)
   // ... calc(i+2)
}
```

CAPS

Unrolling Split Strategy

- Split the iteration space

```
#!hmppcg unroll(4), split
DO i=1,1000
   A(i) = 1
ENDDO
```

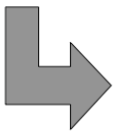
```
DO k=1,250
 A(  0+k) = 1
 A(250+k) = 1
 A(500+k) = 1
 A(750+k) = 1
ENDDO
```

- *Split* is usually recommended to unroll the loop on which the WARPs are mapped
  - Help keeping a good coalescing
- e.g WARP from the purple statement accesses A(501), A(502), A(503), ... A(516)
  - This is contiguous between threads so well coalesced
  - A regular unrolling would do A(3), A(7), A(11), ... , A(59), A(63)

Unroll & Jam

- Unroll & Jam combines blocking and unrolling
  - Can reduce the number of memory accesses
  - Can increase in-register locality
  - But more registers implies less threads/blocks in parallel

```
$!hmppcg unroll(2), jam(1), noremainder
DO i=1,n
  $!hmppcg unroll(2), noremainder
  DO j=1,n
     ... calc(i,j)
  ENDDO
ENDDO
```



```
DO i=1,n,2
   DO j=1,m,2
    ... calc(i+0,j+0)
    ... calc(i+0,j+1)
    ... calc(i+1,j+0)
    ... calc(i+1,j+1)
   ENDDO
ENDDO
```

Unroll & Jam Example

```
#pragma hmppcg unroll(4), jam(2), noremainder
 for( j = 0 ; j < p ; j++ ) {
   #pragma hmppcg unroll(4), split, noremainder
   for( i = 0 ; i < m ; i++ ) {
     double prod = 0.0;
     double v1a,v2a ;
     k=0 ;
     v1a = vin1[k][i] ;
     v2a = vin2[j][k] ;
     for( k = 1 ; k < n ; k++ ) {
       prod += v1a * v2a;
      v1a = vin1[k][i] ;
      v2a = vin2[j][k] ;
     }
     prod += v1a * v2a;
     vout[j][i] = alpha * prod + beta * vout[j][i];
   }
 }
```


Loop Collapsing

- Gather multiple loops into one

```
for (j = 1; j <= lastrow-firstrow+1; j++)
    sum = 0.0;
    for (k = rowstr[j]; k < rowstr[j+1]; k++) {
        sum = sum + a[k]*p[colidx[k]];
    }
   w[j] = sum;
}
```

```
#pragma hmppcg parallel
   for (j = 1; j <= maxrow; j++) {
        tmp_w[j] = a[j]*p[colidx[j]];
   }
 .. do  reduction of the temporary array ..
```

```
DO i=1,n
 DO j=1,m
  DO k=1,p
    ... calc(i,j,k)
  ENDDO
 ENDDO
ENDDO
```

```
DO i=1,n
 DO z=1,m*p
   k=z%p+1
   j=z/p+1
   ... calc(i,j,k)
 ENDDO
ENDDO
```

Exploiting Shared Memory

- Some data are temporarily mapped to the streaming multiprocessors shared memories
- Need to specify a mapping function from a large memory address space to a smaller one
- Strong interaction with unroll (& jam)

```
!$hmppcg block scratch S[BS(i)+2]
do i=2,n-1
!$hmppcg block mapping T1[$1] S[$1%(BS(i)+2)]
!$hmppcg block update S[:] <- T1[i-1:i+1]
     T2(i) = T1(i) + T(i-1) + T(i+1)
!$hmppcg end block mapping T1
enddo
```


Example (DP) of Impact of the Various Tuning Steps

- Original code = 1.0 Gflops
- Improved coalescing (change loop order) = 15.5 Gflops
- Exploit SM shared memories = 39 Gflops
- Better register usage (unroll & jam) = 45.6 Gflops

```
DO j=1+2,n-2
  DO i=1+2,n-2
    DO k=1,10
     B(i,j,k) = &
        & c11*A(i-2,j-2,k) + c21*A(i-1,j-2,k) + c31*A(i+0,j-2,k) + c41*A(i+1,j-2,k) +  c51*A(i+2,j-2,k) + &
        & c12*A(i-2,j-1,k) + c22*A(i-1,j-1,k) + c32*A(i+0,j-1,k) + c42*A(i+1,j-1,k) +  c52*A(i+2,j-1,k) + &
        & c13*A(i-2,j+0,k) + c23*A(i-1,j+0,k) + c33*A(i+0,j+0,k) + c43*A(i+1,j+0,k) +  c53*A(i+2,j+0,k) + &
        & c14*A(i-2,j+1,k) + c24*A(i-1,j+1,k) + c34*A(i+0,j+1,k) + c44*A(i+1,j+1,k) +  c54*A(i+2,j+1,k) + &
        & c15*A(i-2,j+2,k) + c25*A(i-1,j+2,k) + c35*A(i+0,j+2,k) + c45*A(i+1,j+2,k) +  c55*A(i+2,j+2,k)
    ENDDO
  END DO
END DO
```

Fallback Code Issue

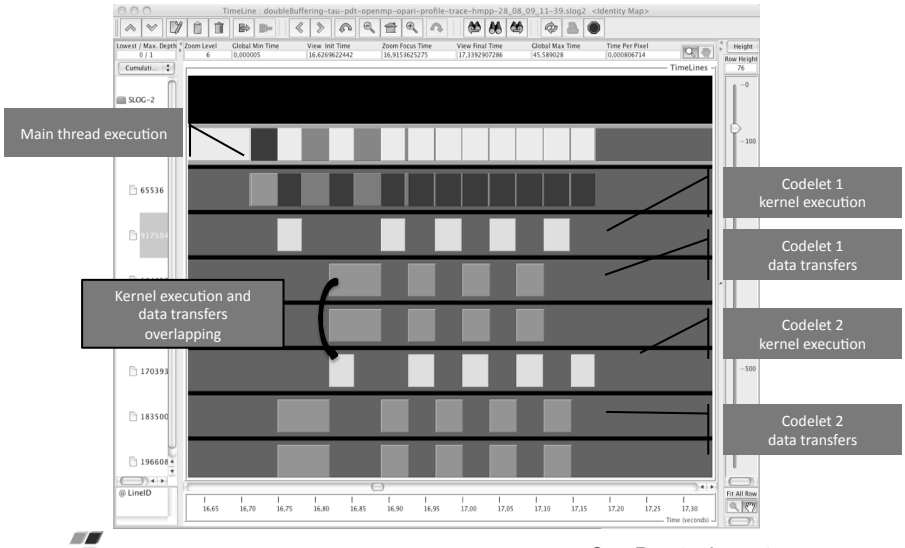
- Fallback codelet is used when the GPU is unavailable
  - Loop transformations for GPU are usually not adequate for CPU performance
- GPU specific optimizations should not degrade fallback codelets
  1. Duplicate the code, one for CPU, the other for GPU
  2. Use guarded pragma
     - `#pragma hmppcg(CUDA) unroll, …`

Miscellaneous

- Other useful loop transformations
  - Blocking
  - Distribution and fusion
  - Software pipelining
  - …
- Global operations
  - Reductions and Barriers
- Avoiding control flow divergence
- Data access alignment
- …

Dealing with CPU-GPU Communication Tuning: HMPP-TAU

Double buffering example

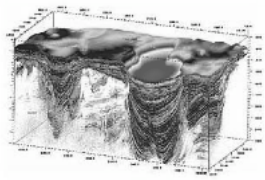


 See Paratools poster

Application Example (RTM)

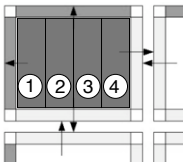
Seismic Modeling Application

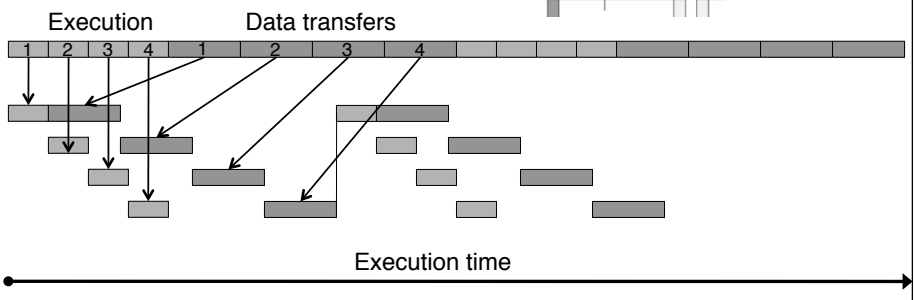
- Reverse Time Migration modeling at Total
  - Acceleration of critical functions
  - Use HMPP with CUDA
- Data domain decomposition
  - Large data processing
  - One sub-domain running on a node with two GPU cards
- Main issue
  - Optimization of communications between CPUs and GPUs
- Results (Q2/09)
  - 1 GPU-accelerated machine is equivalent to 4.4 CPU machines
  - GPU: 16 dual socket quadcore Hapertown nodes connected to 32 GPUs
  - CPU: 64 dual socket quadcore Hapertown nodes



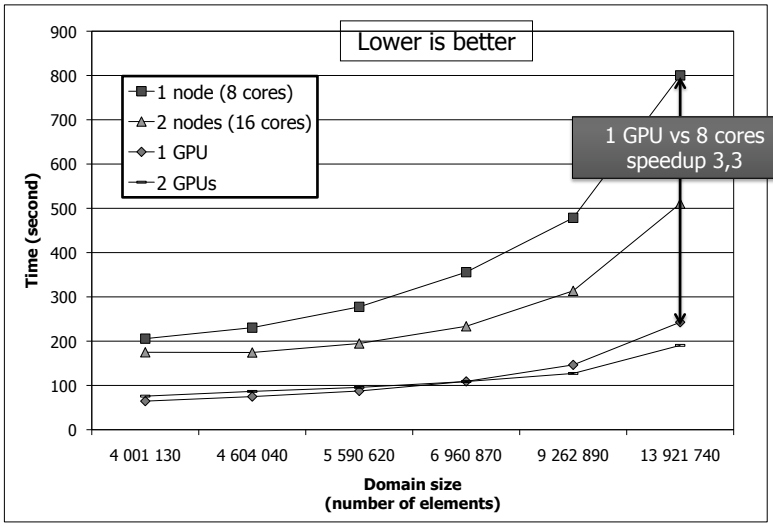
Overlapping Kernel Execution with Data Transfers

- Use asynchronous data transfers between CPU and GPU
  - Divide sub-domain computations in streams
  - Use partial data transfers





CPU Versus GPU (Domain size varies)



Conclusion

Conclusion

- High level GPU code generation allows many GPU code optimization opportunities
  - Thread layout and thread body need to be considered together
- Hardware cannot be totally hidden to programmers
  - e.g. exposed memory hierarchy
  - Efficient programming rules must be clearly stated
- Quantitative decisions as important as parallel programming
  - Performance is about quantity
  - Tuning is specific to a GPU configuration
  - Runtime adaptation is a key feature
    - Algorithm, implementation choice
    - Programming/computing decision

CAPS

Innovative Software for Manycore Paradigms