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
Manycore Architectures

- General purpose cores
  - Share a main memory
  - Core ISA provides fast SIMD instructions

- Streaming engines / DSP / FPGA
  - Application specific architectures ("narrow band")
  - Vector/SIMD
  - Can be extremely fast

- Hundreds of GigaOps
  - But not easy to take advantage of
  - One platform type cannot satisfy everyone

- Operation/Watt is the efficiency scale
  - e.g. one SGI ICE: 64 CPU nodes (22 kW) vs one SGI ICE GPGPU: 8 Tesla S1070 (32 GPUs) + 16 CPU nodes (12 kW)
Stream Computing

- A similar computation is performed on a collection of data (stream)
  - There is no data dependence between the computation on different stream elements
Consequences of Manycores in Main Stream

• Require more programming expertise
  o Manycore is not a market request but it cannot be avoided

• Adds uncertainty to programming development technology

• Need to consider questions such as
  o Is it possible to make a single (a few) binary that runs efficiently for a large set of hardware configurations?
  o How to deal with non homogeneous core behavior?
  o How to organize the compilation flow?
  o How to ensure that conflicts in resource usage will not lead to global performance degradation?
  o How to easily debug heterogeneous codes?

• Training and software tools are necessary
  o GPUs are not usual processors (less forgiving)
Outline

1. A short introduction to GPU architecture

2. Overview of HMPP

3. High level code tuning

4. An example of application
A Short Introduction to GPU Architecture
GPU Architecture Overview

• Fine grain massive data parallelism
  o Many streaming processors

• Exposed memory hierarchy
  o Large device memory
  o Small partially *shared* memory, …

• Heavily pipelined memory
  o Small data caches

• Fixed amount of hardware resources
  o No virtual memory
  o Maximum number of threads
  o Maximum number of registers usage per group of threads
  o …
Example: NVIDIA T10

- **Main Memory**
- **Thread Execution Manager**
- **PCle Interface**
- **Host**

**Thread Execution Manager**

- **SM**
  - **SP**
  - Shared memory
  - **SM**
  - **SP**
  - Shared memory
  - **SM**
  - **SP**
  - Shared memory
  - **SM**
  - **SP**
  - Shared memory
  - **SM**
  - **SP**
  - Shared memory

**512-bit memory interconnect**

- **L2**
- **DRAM**
- **DRAM**
- **DRAM**
- **DRAM**
- **DRAM**
- **DRAM**

**GPU Code**

- **Thread Execution Manager**

**PCIe Interface**

**Host**

**Streaming Multiprocessor (30 units)**

**Streaming Processor (240 units)**

**16kB**

**1 TFlops SP**

**86.4 GFlops DP**

**at 1.3 Ghz**

**102 GB/s bandwidth**

www.caps-entreprise.com
CUDA (OpenCL) Grids and Blocks

- GPUs need 1000s of threads to be efficient
  - Highly pipelined
  - Highly parallel

- Blocks of thread are executed on the streaming multiprocessors
Consequences of GPU Architectures

• Host-Device model
  o High overhead when moving data between CPU and GPU, not always efficient to move a computation to GPU

• Memory bound device
  • High arithmetic computing density needed
  o Spatial locality between threads is crucial
    • Exploit memory access coalescing capabilities

• Sensible to Resource allocation
  o Large number of registers: loop unrolling is an important transformation
  o Small streaming multiprocessor shared memory: help reducing pressure on device memory

**Penalty for over-spending a type of resource is huge**
  • e.g. too many threads by block
  • e.g. too many registers per thread, spilling is very expensive

• Expensive global synchronization
  o No global hardware thread communication

• Performance is very problem size dependent
  o The number of threads and complexity of threads depend on the number of computations to perform
What’s New in the Landscape of GPU computing

• OpenCL from Khronos consortium
  o C based
  o Low level, great for code generation tool
  o http://www.khronos.org/

• NVIDIA Fermi architecture
  o Enhanced double precision architecture
  o ECC
  o Etc.
HMPP Overview
from the Application to the Device
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
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
  o Do not require a new programming language
    • And can be applied to many based languages
  o Already state of the art approach (e.g. OpenMP)
  o Keep incremental development possible

• Portability
  o To various hardware accelerators (HWAs) and host
code is independent of the HWA

• Avoid exit cost
• Bring remote procedure call on GPU
  o Host & compute device model
  o Synchronous and asynchronous

➢ Allow to optimize communication between CPU and GPU

• Provide high level GPU code generation from sequential code

➢ Provide device resource management
#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 );
}
int main(int argc, char **argv) {
    
    #pragma hmpp sgemm allocate, args[vin1;vin2;vout].size={size,size}
    
    do something
    
    #pragma hmpp sgemm callsite, asynchronous
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    do something else
    
    #pragma hmpp sgemm synchronize
    #pragma hmpp sgemm delegatedstore, args[vout]
    do something else
    #pragma hmpp sgemm release

    Allocate and initialize device early
    Execute asynchronously
    Download result when needed
    Release HWA
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
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
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
  o Allow to exploit reuses of argument to reduce communications
  o Close to equivalence in Fortran

```c
#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

```c
#pragma hmpp region
{
  int i;
  for (i = 0 ; i < n ; i++) {
    v1[i] = v2[i] + v3[i]*alpha;
  }
}
```
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]
}
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, C and FORTRAN
- Seamless integration of GPU libraries such as cuFFT and cuBLAS
- GPU code tuning directives
HMPP Performance
High Level GPU Code Tuning
High Level GPU Code Tuning

• High level GPU code generation aims at removing low-level implementation details
  o But users still need to understand the underlying process

• Two different sets of tuning issues
  o Global to application: CPU-GPU data transfers
  o Local to threads: GPU kernels

• Many parameters to deal with when exploring the thread optimization space
  o Easier to explore at higher code level
  o Many useful loop transformations
  o Interaction with thread layout on the streaming multiprocessors
High Level Code Generation

Codelet

Parallel loop nests

Group of threads + Threads mapping

Compiler options

GPU Compiler

GPU Code

Compiler options

GPU Compiler

GPU Code

Compiler options

GPU Compiler

GPU Code

Compiler options

GPU Compiler

GPU Code
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

```fortran
!$hmppcg parallel
DO i=0,n
    !$hmppcg noparallel
    DO j=1,4
        A(j,i) = . . .
    ENDDO
ENDDO
```
for (i = 1; i < m-1; ++i){
    for (j = 1; j < n-1; ++j) {
                  +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];
    }
}
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

```fortran
DO j=1,m    ! 2^{nd} grid dimension
  DO i=1,n  ! 1^{st} 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 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)
A Small Experiment with Loop Permutations

- 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

<table>
<thead>
<tr>
<th></th>
<th>JIK</th>
<th>JKI</th>
<th>IJK</th>
<th>IKJ</th>
<th>KJI</th>
<th>KIJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min Perf</td>
<td>0,8</td>
<td>5</td>
<td>0,1</td>
<td>10</td>
<td>0,1</td>
<td>0,8</td>
</tr>
<tr>
<td>Max Perf</td>
<td>1</td>
<td>14,4</td>
<td>0,2</td>
<td>15,9</td>
<td>0,5</td>
<td>3</td>
</tr>
</tbody>
</table>

```c
#include <stdio.h>

#define M 1024
#define N 1024
#define P 128

int main() {
    float A[M][N][P], B[M][N][P], c11, c12, c21, c22;
    int i, j, k;

    // Initialization of A and B

    // 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] + ...;
            }
        }
    }
    return 0;
}
```
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
  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
```
#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];
    }
}
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)

```c
!$hmppcg block scratch S[BS(i)+2]
do i=2,n-1
  !$hmppcg block mapping T1[$1] S[$1%(BS(i)+2)]
  !$hmppcg block update S[: ] <- T1[i-1:i+1]
    T2(i) = T1(i) + T1(i-1) + T1(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
  ENDDO
ENDDO
END DO
END DO
```
Miscellaneous

• Other useful loop transformations
  o Loop coalescing
  o Blocking
  o Distribution and fusion
  o Software pipelining
  o …

• Global operations
  o Reductions and Barriers

• Avoiding control flow divergence
• Data access alignment
• …
Dealing with CPU-GPU Communication Tuning: HMPP-TAU

Double buffering example

- Main thread execution
- Kernel execution and data transfers overlapping
- Codelet 1 kernel execution
- Codelet 1 data transfers
- Codelet 2 kernel execution
- Codelet 2 data transfers

www.caps-entreprise.com
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)

Lower is better

1 GPU vs 8 cores speedup 3.3
Conclusion
Conclusion

• High level GPU code generation allows many GPU code optimization opportunities
  o Thread layout and thread body need to be considered together

• Hardware cannot be totally hidden to programmers
  o e.g. exposed memory hierarchy
  o Efficient programming rules must be clearly stated

• Quantitative decisions as important as parallel programming
  o Performance is about quantity
  o Tuning is specific to a GPU configuration
  o Runtime adaptation is a key feature
    • Algorithm, implementation choice
    • Programming/computing decision
Innovative Software for Manycore Paradigms