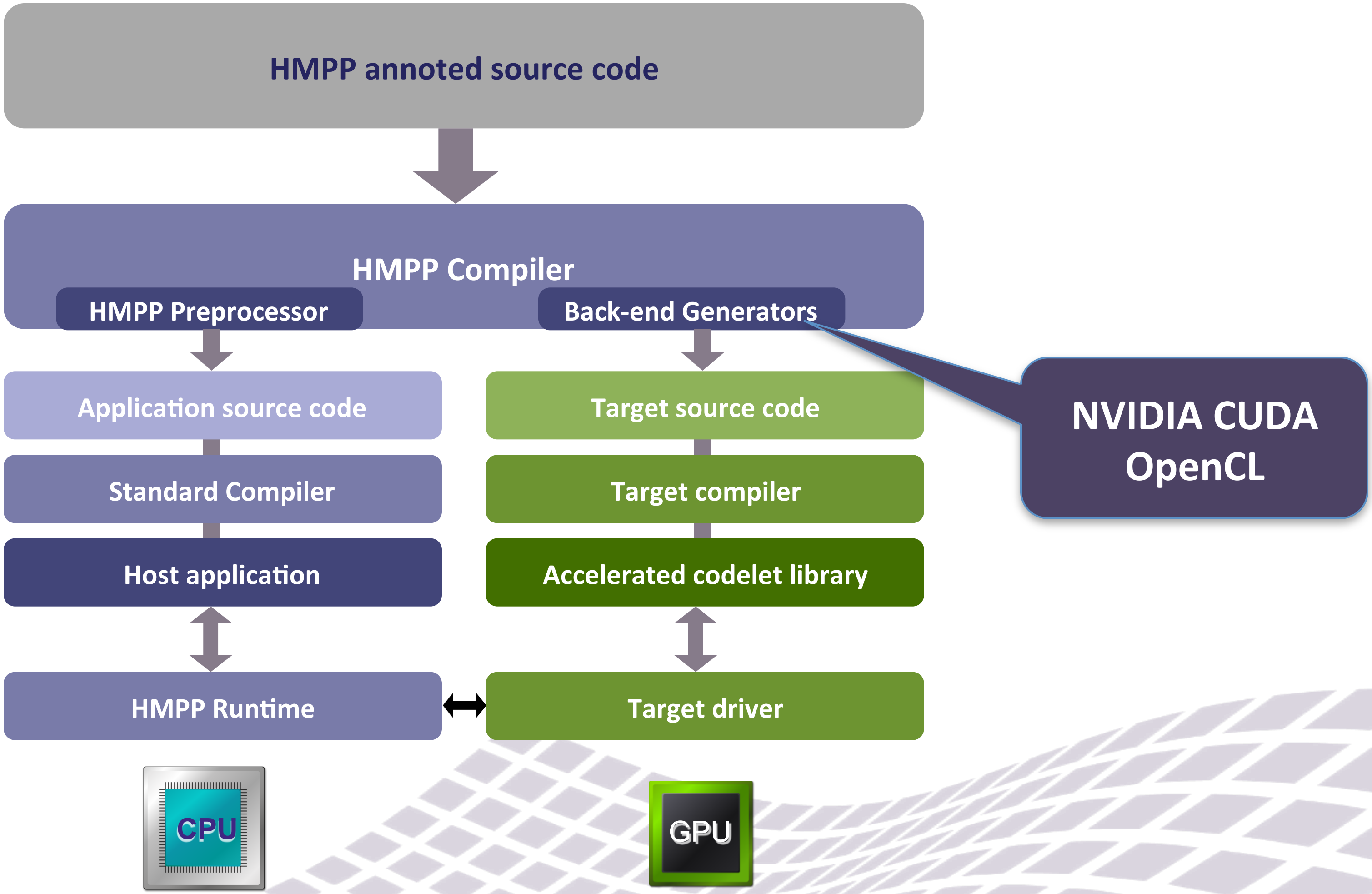
HMPP
Directive Programming

Compilation Workflow



Methodology for Porting Code With HMPP

1. Profiling to identify hotspot(s).
2. Basic offloading with codelet/callsite directives.
3. Avoid transfers by specifying function argument intents.
4. Avoid needless (de)allocations of the HWA with allocate/release directives.
5. Let HMPP automatically optimize the data transfer
6. Preload data and let them stay on the HWA if possible to avoid needless transfers.
7. Launch the codelet asynchronously (allow the host to execute other tasks while the HWA executes the codelet).
8. Gather several hotspots by creating codelet groups.
9. Share data between codelets with data mirroring, mapping and resident data.
10. Transfer only the useful part of your data/result
11. Compute automatically on multiple devices

Test Platform

- All performance measures presented in this training are made on a machine with the following characteristics:
  - HWA:
    - 1 x nVidia Tesla C2050
    - 3 Gb RAM
    - 14 multiprocessors
    - 448 cores
    - 1.15 GHz
    - CUDA 4.1
  - Host:
    - Intel(R) Core i7 @ 2.67GHz
    - 6 GB DDRIII
    - HMPP 3.0.5
    - Gcc 4.4.5

Lab 0 - Profiling to Identify Hotspots(s)

- To have a worthwhile acceleration by offloading computations on a HWA, you need to move the most expensive computations (aka hotspots) on it (Amdahl's law)
- You can identify hotspots with profilers such as Gprof or Oprofile
- You may have to create codelet functions or to use HMPP regions to explicitly create the part of code which will be offloaded
- $C = \alpha AB + \beta C$
- Profiling output of lab0 gives:

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
100.29	**41.46**	**41.46**	**5**	**8.29**	**8.29**	**mySgemm**
0.05	41.46	0.02	3145729	0.00	0.00	randFloat

- It's clear that this application's hotspot is "mySgemm"

Define a Codelet Function

- Declare a hardware-accelerated version of a function:
  - Use the codelet/callsite pair of directives

```
#pragma hmpp mySgemm codelet, target=CUDA, args[*].transfer=atcall
void mySgemm( int m, int n, int k, float alpha, float beta,
              float a[m][n], float b[n][k], float c[m][k] )
{ /* . . . */ }

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

  /* . . . */

  for( j = 0 ; j < NB_RUNS ; j++ ) {
    /* . . . */
    #pragma hmpp mySgemm callsite
    mySgemm( m, n, k, alpha, beta, a, b, c_hwa );
    /* . . . */
  }
}
```

Declare codelet with CUDA target

Synchronous codelet call

  - The HWA will be automatically allocated
    - Data transfers between the host and the HWA will be handled transparently prior to the codelet's execution due to the transfer policy ATCALL

Hands-on Lab 1

- See HMPP User Manual
  - Section 3.1 – The HMPP Codelet Concept for a more complete definition of what a codelet is
  - Section 4.3 – Syntax of the HMPP directives for more information on how to read and write HMPP directives
  - Section 4.5.3 – callsite directive for the callsite definition
  - Section 4.5.1 – codelet directive for the codelet definition
  - Section 4.7.1 – ATCALL transfer policy

- Compile and run example lab1
- What to look for
  - Note the command line used for the generation of the application,
  - Note the generation of the codelet
  - Note the speed-up between the code that runs on the CPU and on the HWA

- Add runtime information by exporting environment variable HMPPRT_LOG_LEVEL=INFO, and note the HMPP runtime mechanisms
  1. Allocation of the HWA
  2. Data transfer write
  3. Execution of the codelet
  4. Data transfer read
  5. Release of the HWA

Speed-up HMPP/ CUDA VS CPU	
215x	

Avoid Some Needless Transfers By Specifying Intents

- Scalar variables are all IN by default
- Arrays are all INOUT by default
  - Arrays as INOUT wastes transfer bandwidth
  - But “const” arrays automatically have an IN intent

```
#pragma hmpp mySgemm codelet, target=CUDA, args[*].transfer=atcall
void mySgemm( int m, int n, int k, float alpha, float beta,
              const  float a[m][n], const  float b[n][k], float c[m][k] )
{ /* . . . */ }

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

 for( j = 0 ; j < NB_RUNS ; j++ ) {
#pragma hmpp mySgemm callsite
    mySgemm( m, n, k, alpha, beta, a, b, c_hwa );
  }
    /* . . . */
 }
```

Only “c” is INOUT

Hands-on Lab 2

- See HMPP User Manual
  - Section 4.5.1 – codelet directive
- Compile and run example lab2
- What to look for
  - Note that the speed-up between the code that runs on the CPU and on the HWA has improved

Speed-up HMPP/ CUDA VS CPU	Speed-up VS previous version
248x	1.15x

- Increase the HMPP log level by exporting environment variable HMPPRT_LOG_LEVEL=INFO, and note that less “upload” transfers are generated

Allocate and Release

- Explicitly acquire the device and allocate the data on the device before the callsite
  - Use the acquire/allocate
  - Explicit sizes of pointers arguments
  - Device release done implicitly at the program ending

```
int main(int argc, char **argv) {
  /* . . . */
  #pragma hmpp mySgemm acquire

  #pragma hmpp mySgemm allocate, args[a].size={m,n}, args[b].size={n,k},
args[c].size={m,k}

    /* . . . */

  for( j = 0 ; j < 2 ; j++ ) {
    /* . . . */
    #pragma hmpp mySgemm callsite
    mySgemm( m, n, k, alpha, beta, a, b, c_hwa );
    /* . . . */
  }
  /* . . . */
}
```

Acquire explicitly a device

Pre-Allocate data on the device

Hands-on Lab 3

CAPS

- See HMPP User Manual
  - Section 4.5.5 – acquire Directive
  - Section 4.5.7 – allocate Directive

- Compile and run example lab3
- What to look for
  - Note that the speed-up between the code that runs on the CPU and on the HWA is the same

Speed-up HMPP/ CUDA VS CPU	Speed-up VS previous version
248x	1,00x

- Increase the HMPP log level by exporting environment variable HMPPRT_LOG_LEVEL=INFO, and note
  - The allocation of the HWA is only done once
  - Data transfers write/read, codelet execution remain unchanged
  - The HWA is released automatically at the end of the application

HMPP Automatically Optimize and Factorize Data Transfers (Experimental)

CAPS

- Data preloading is done on first callsite
  - Synchronization between host and GPU is then done only when HMPP detects a modification of data (hostread/hostwrite in log)
  - Avoid useless transfers with pragma disregard

```
#pragma hmpp mySgemm codelet, target=CUDA, args[*].transfer=auto
void mySgemm( int m, int n, int k, float alpha, float beta,
              const  float a[m][n], const  float b[n][k], float c[m][k] )
{ /* . . . */
}

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

  #pragma hmpp mySgemm disregard args[*]
  for( j = 0 ; j < NB_RUNS ; j++ ) {
    /* . . . */
    #pragma hmpp mySgemm callsite
    mySgemm( m, n, k, alpha, beta, a, b, c_hwa );
  }
  /* . . . */
}
```

Let HMPP determine when to transfer data

Let HMPP determine when to transfer data

Hands-on Lab 4

- See HMPP User Manual
  - Section 4.7.4 - Automatic data transfer in HMPP – .transfer=auto
- Compile and run example lab4
- What to look for
  - Note that the speed-up between the code that runs on the CPU and on the HWA has improved

Speed-up HMPP/ CUDA VS CPU	Speed-up VS previous version
366x	1.48x

- Increase the HMPP log level by exporting environment variable HMPPRT_LOG_LEVEL=INFO, and note that the number of transfers is reduced.
- Remove the disregard args[*] clause and note that the number of transfers has increased
  - That clause instructs HMPP that calls to intrinsics won't have side-effects on variables used into the codelet. Without this, HMPP suppose they may be and then transfer data onto the HWA at each iteration.
  - This increase the CPU time execution due to the host variable access analysis

Manually Optimize and Factorize Data Transfers

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

```
#pragma hmpp mySgemm codelet, args[m;n;k;alpha;beta;a;b].transfer=atfirstcall, &
#pragma hmpp & args[c].transfer=manual, args[c].mirror, target=CUDA
void mySgemm( int m, int n, int k, float alpha, float beta,
              const  float a[m][n], const  float b[n][k], float c[m][k] ){}

int main(int argc, char **argv) {
  /* . . . */
  #pragma hmpp sgemm allocate, args[a].size={m,n}, args[b].size={n,k}, args[c].size={m,k}
  /* . . . */
  #pragma hmpp mySgemm advancedload, args[c], hostdata="c_hwa"
  /* . . . */

  for( j = 0 ; j < NB_RUN ; j++ ) {
    #pragma hmpp mySgemm callsite
    mySgemm( m, n, k, alpha, beta, a, b, c_hwa );
    /* . . . */
  }
  #pragma hmpp mySgemm delegatedstore, args[c]
  /* . . . */
```

Avoid reloading data

Manage Manually c

Preload data

Download result

Hands-on Lab 5

- See HMPP User Manual
  - Section 4.6.1 – advancedload Directive
  - Section 4.6.2 – delegatedstore Directive
  - Section 4.7.3 – MANUAL transfer policy

- Compile and run example lab5

- What to look for
  - Note that the speed-up between the code that runs on the CPU and on the HWA is the same

Speed-up HMPP/ CUDA VS CPU	Speed-up VS previous version
366x	1,00x

- Increase the HMPP log level by exporting environment variable HMPPRT_LOG_LEVEL=INFO, and note that the number of transfers is reduced
  - See the directive Advanceload and in the first callsite in the log. Some arguments are only transferred at the first callsite.

Compute Asynchronously

- Perform CPU/GPU computations asynchronously
  - Asynchronous execution allows to perform other computations on the host while the codelet is executed on the HWA, or to avoid needless synchronizations between the HWA and the host

```
#pragma hmpp mySgemm codelet, target=CUDA, args[m;n;k;alpha;beta].transfer=atfirstcall, args
[a,b,c].transfer=manual, args[a,b,c].mirror
void mySgemm( int m, int n, int k, float alpha, float beta,
              const  float a[m][n], const  float b[n][k], float c[m][k]

int main(int argc, char **argv) {
    /* . . . */
#pragma hmpp mySgemm allocate, args[a].size={m,n}, args[b].size={n,k}, args[c].size={m,k}
#pragma hmpp mySgemm advancedload, args[c], hostdata="c_hwa"
    /* . . . */
 for( j = 0 ; j < 2 ; j++ ) {
#pragma hmpp mySgemm callsite asynchronous
  mySgemm( m, n, k, alpha, beta, a, b, c_hwa );
    /* . . . */
  }
    /* . . . */

#pragma hmpp mySgemm synchronize
#pragma hmpp mySgemm delegatedstore, args[c]
}
```

Execute asynchronously

Wait for codelet completion

Download result when needed

Hands-on Lab 6

- See HMPP User Manual
  - Section 4.5.3 – asynchronous callsite Directive
  - Section 4.5.4 – synchronize Directive
- Compile and run example lab6
- What to look for
  - Note that the speed-up between the code that runs on the CPU and on the HWA is the same

Speed-up HMPP/ CUDA VS CPU	Speed-up VS previous version
325x	1.00x

- Increase the HMPP log level by exporting environment variable HMPPRT_LOG_LEVEL=INFO, and note that the codelet is run asynchronously.

What Can CAPS entreprise Do with You?

CAPS Compute Lab

Evaluate the performance of your application on a ready-to-use hybrid cluster

- System key features
  - 42 TFLOPS Tesla accelerated Nehalem cluster
  - 10 NVIDIA Tesla S1070
  - 20 Intel Nehalem nodes

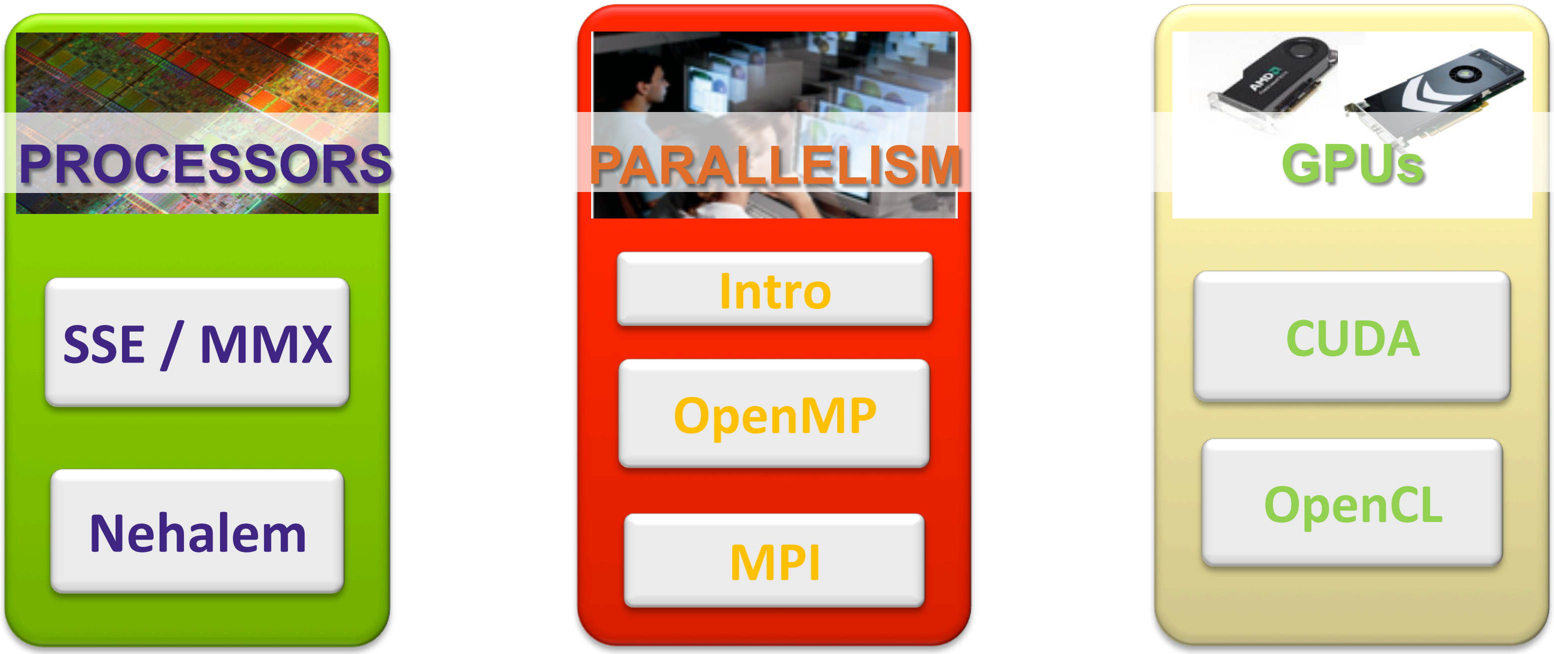
- Complete software environment
  - HMPP, CUDA, Intel compilers
- CAPS expertise and consultancy

CAPS Expertise

CAPS provides consulting to

- Develop, port and deploy applications onto hybrid systems
- Optimize GPU-accelerated applications
- Bring our expertise to your team

CAPS Training



- See CAPS web site for full program
- or SGI partner web site (www.hpctraining.com)

Accelerator Programming Model
Parallelization
CAPS
Directive-based programming
GPGPU
Manycore programming
Hybrid Manycore Programming
HPC community
Petaflops
Parallel computing
HPC open standard
Multicore programming
Exaflops
NVIDIA Cuda
Code speedup
Hardware accelerators programming
High Performance Computing
Parallel programming interface
Massively parallel
Open CL

CAPS
open hmpp

http://www.caps-entreprise.com
http://twitter.com/CAPSentreprise

http://www.openhmpp.org