CAPS

Many-Core Programming with HMPP 3.0

Write once, deploy many(-core)

Jean-Charles VASNIER

Introduction

- Computing power comes from parallelism
  - Hardware (frequency increase) to software (parallel codes) shift
  - Driven by energy consumption : heterogeneity is source of efficiency

- Context of fast moving hardware targets
  - e.g. fast GPU improvements (RT and HW), new massively parallel CPU
  - Write codes that will last many architecture generations

- Keeping a unique version of the code, preferably mono-language, is a necessity
  - Reduce maintenance cost
  - Directive-based approaches suitable
  - Preserve code assets

- Addressing many-core programming challenge implies
  - Massively parallel algorithms
  - New development methodologies / code architectures
  - New programming tools

Agenda

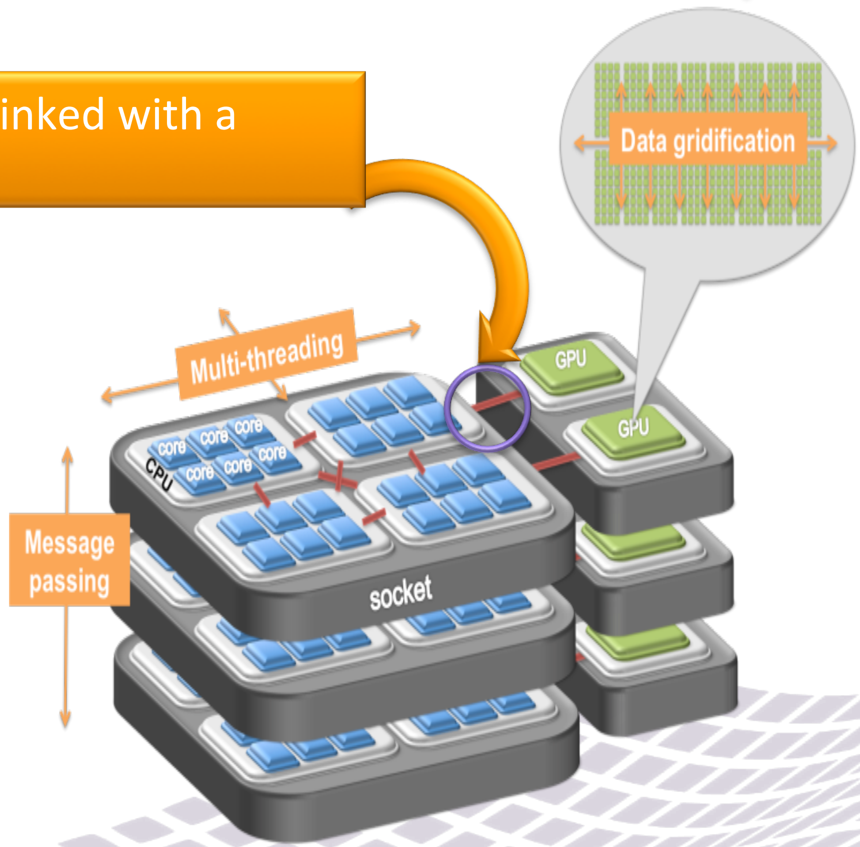
1. About many-cores
2. Some principles for many-core programming
3. Methodology to migrate legacy codes
4. Many-core programming with HMPP 3.0
5. HMPP Wizard

Many-Cores

CAPS

- Massively parallel devices
  - Thousands of threads needed

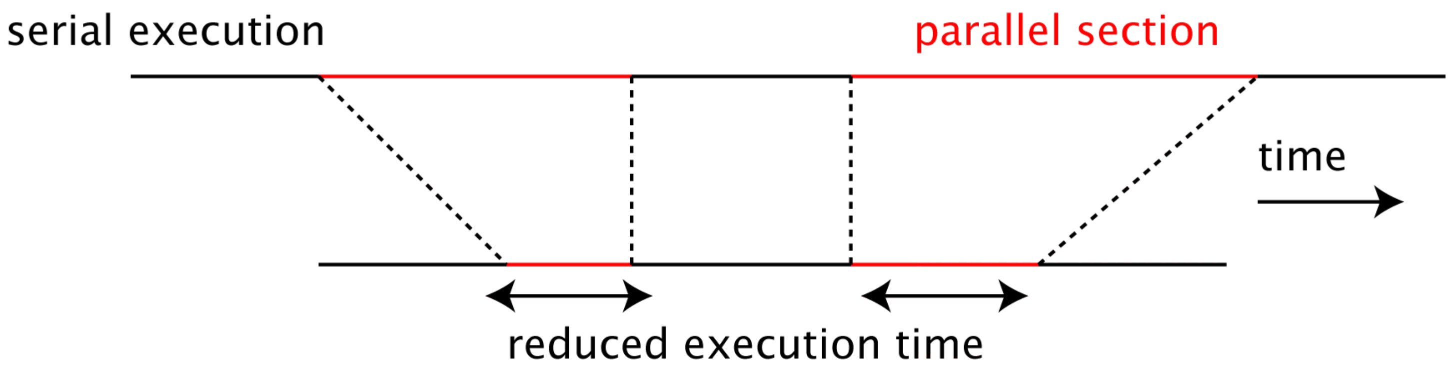
CPU and GPUs linked with a PCIx bus



Data/stream/vector parallelism to be exploited by GPUs
e.g. CUDA / OpenCL

Software Main Driving Forces

- ALF - Amdahl's Law is Forever
  - A high percentage of the execution time has to be parallel
  - Many algorithms/methods/techniques will have to be reviewed to scale



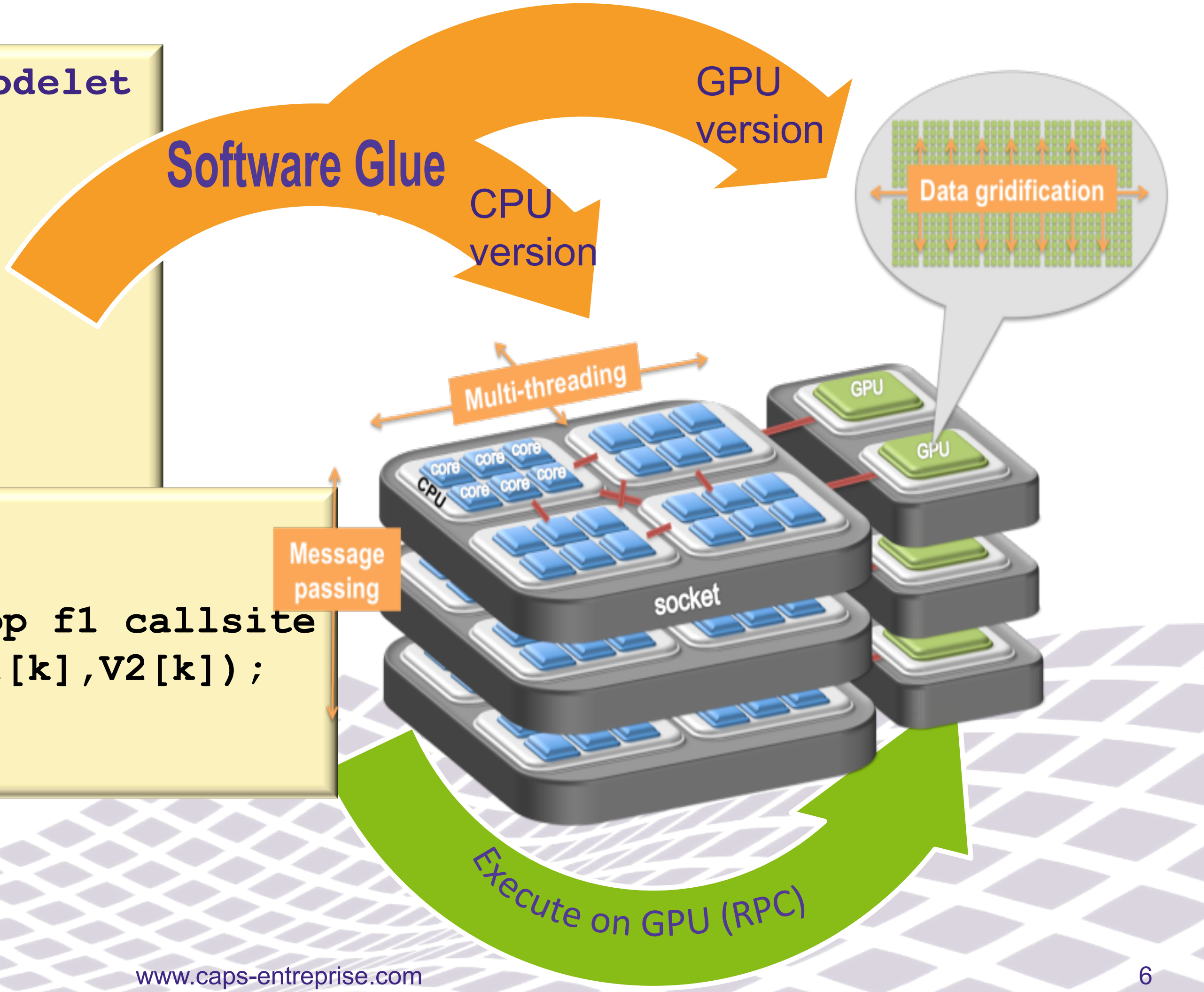
- Data locality is expected to be the main issue
  - Limit on the speed of light
  - Moving data will always suffer latency

Addressing many-cores

- A directive-based approach for many-core
  - CUDA, OpenCL and soon Intel MIC, …

```
#pragma hmpp f1 codelet
myfunc(...){
 ...
 for()
  for()
   for()
    ...
 ...
}
```

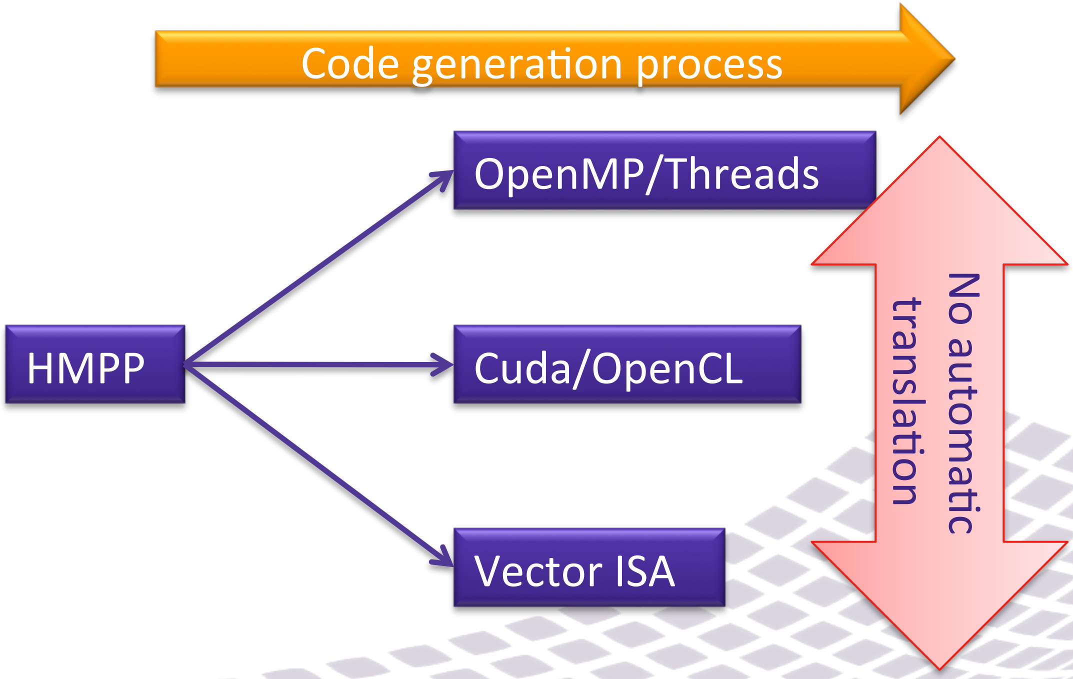
```
main(){
 ...
#pragma hmpp f1 callsite
  myfunc(V1[k],V2[k]);
 ...
}
```



Some Principles for
Many-Core Programming

Express Parallelism, not Implementation

- Rely on code generation for implementation details
  - Usually not easy to go from a low level API to another low level one
  - Tuning has to be possible from the high level
  - But avoid relying on compiler advanced techniques for parallelism discovery, …
  - **You may have to change the algorithm!**
- An example with HMPP




```
#pragma hmppcg gridify(j,i)
#pragma hmppcg unroll(4), jam(2)
  for( j = 0 ; j < p ; j++ ) {
    for( i = 0 ; i < m ; i++ ) {
     for (k = ...) { ...}
      vout[j][i] = alpha * ...;
    }
  }
```

Exposing Massive Parallelism

- Do not hide parallelism with complex coding structure
  - Data structure aliasing, …
  - Deep routine calling sequences
  - Separate concerns (functionality coding versus performance coding)

- Data parallelism when possible
  - Simple form of parallelism, easy to manage
  - Favor data locality
  - But sometimes too static

- Kernels level
  - Expose massive parallelism
  - Ensure that data affinity can be controlled
  - Make sure it is easy to tune the ratio vector / thread parallelism

Data Structure Management

- Data locality
  - Makes easy to move from one address space to another one
  - Makes easy to keep data coherency
- Do not waste memory
  - Memory per core ratio is not improving
- Choose simple data structures
  - Enable vector/SIMD computing
  - Use library friendly data structures
  - May come in multiple forms, e.g. sparse matrix representation
- For instance consider “data collections” to deal with multiple address spaces or multiple devices or parts of a device
  - Gives a level of adaptation for dealing with heterogeneity
  - Load distribution over the different devices is simple to express

Debugging Issues

- Keep code debug-able.

- Keep serial semantic
  - For instance, implies keeping serial libraries in the application code
  - Directives-based programming makes this easy

- Ensure validation is possible even with rounding errors
  - Reductions, …
  - Aggressive compiler optimizations

- Use defensive coding practices
  - Events logging, parameterize parallelism, add synchronization points, …
  - Use debuggers (e.g. Allinea DDT)

Dealing with Libraries

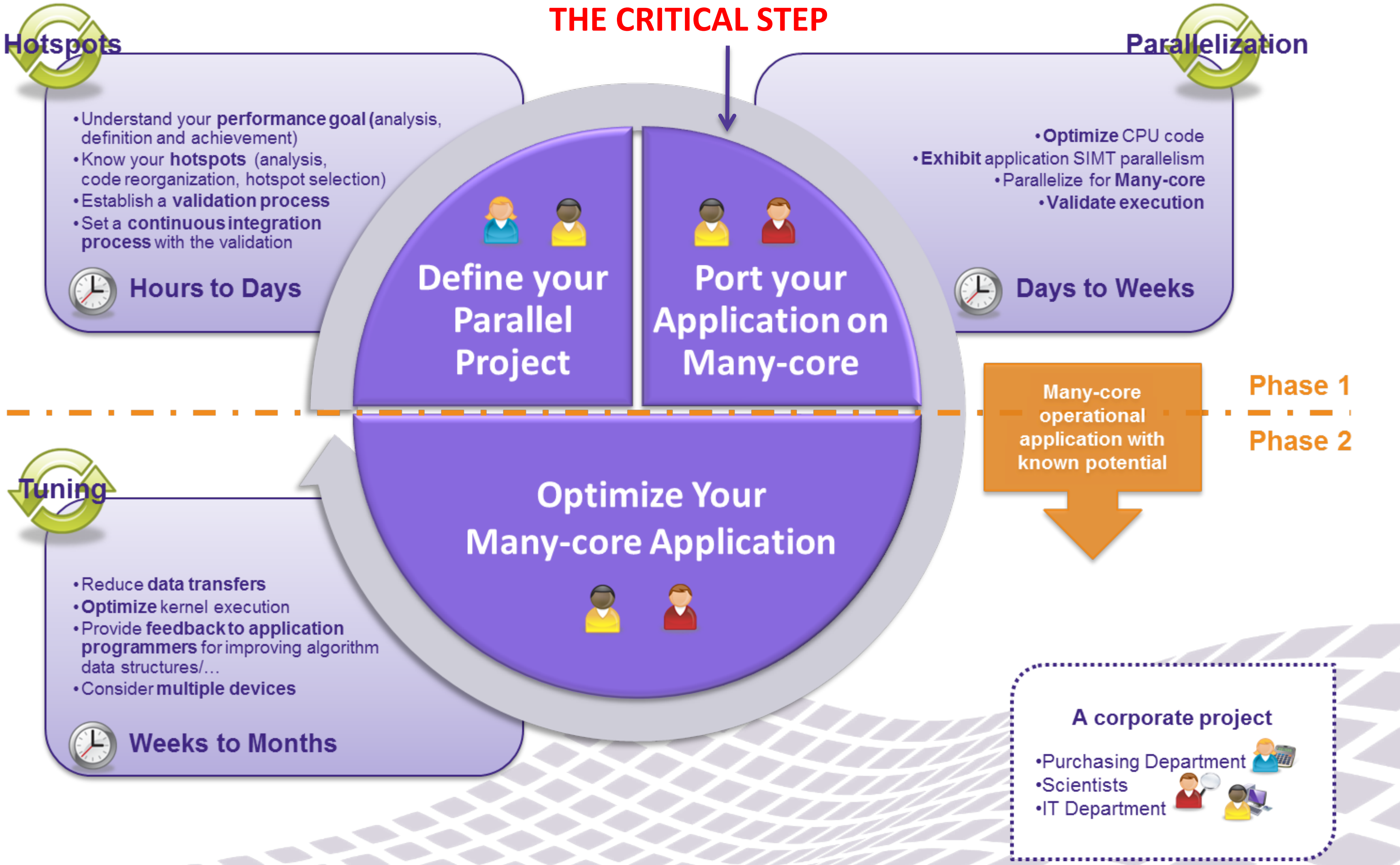
- Library calls can usually only be partially replaced
  - No one to one mapping between libraries (e.g.BLAS, FFTW, CuFFT, CULA, LibJacket)
  - No access to all code (i.e. avoid side effects)
  - Don't create dependencies on a specific target library as much as possible
  - **Still want a unique source code**
- Deal with multiple address spaces / multi-GPU
  - Data location may not be unique (copies)
  - Usual library calls assume shared memory
  - Library efficiency depends on updated data location (long term effect)
- Libraries can be written in many different languages
  - CUDA, OpenCL, HMPP, etc.
- There is not one binding choice depending on applications/users
  - Binding needs to adapt to uses depending on how it interacts with the remainder of the code
  - Choices depend on development methodology

A Methodology for Legacy Code Migration

Legacy Codes Migration Challenges

- Mastering migration cost
  - Ensuring an adequate return on investment
  - Minimizing risk as well as manpower
- Producing code that will last many architecture generations
  - It is safe to assume that the node architecture may change with the renewal of the computer
- Writing developer friendly code
  - Application developers may not be multicore / accelerator / parallelism savvy
  - Once ported, the application still needs to evolve
- Keeping a unique version, preferably mono-language, of the codes
  - Reduce maintenance cost
- Able to use libraries
  - No one-to-one replacement (e.g. FFT libraries)
  - Must interact with non library accelerated kernels

Dealing with Legacy Codes



Go / No Go for GPU Target

Go

- Dense hotspots
- Fast kernels
- Low CPU-GPU data transfers
- Prepare to manycore parallelism

No Go

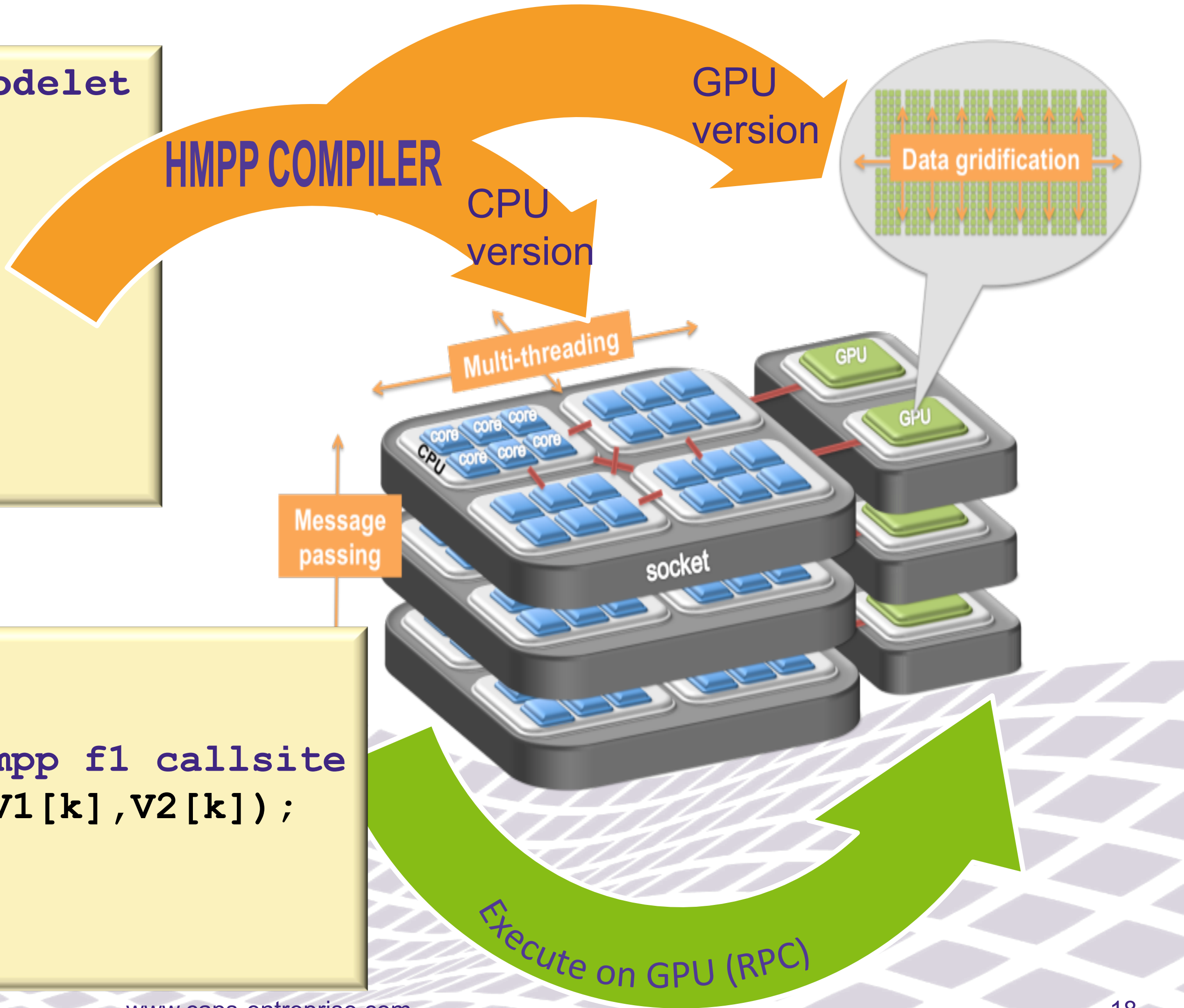
- Flat profile
- Slow GPU kernels (i.e. no speedup to be expected)
- Binary exact CPU-GPU results (cannot validate execution)
- Memory space needed

Many-Core Programming with HMPP 3.0

Scope of HMPP 3.0 Programming

- Remote procedure calls (RPCs) on accelerator devices
  - Parallel loop nests to exploit multiple compute units

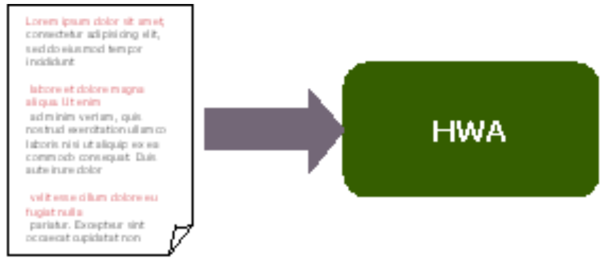
```
#pragma hmpp f1 codelet
myfunc(...){
 ...
 for()
  for()
   for()
    ...
 ...
}
```



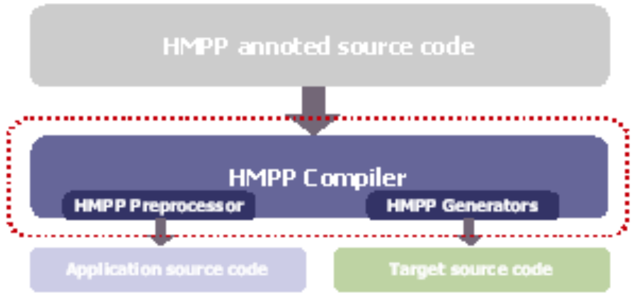

```
main(){
 ...
#pragma hmpp f1 callsite
  myfunc(V1[k],V2[k]);
 ...
}
```

HMPP Comes in 3 Parts

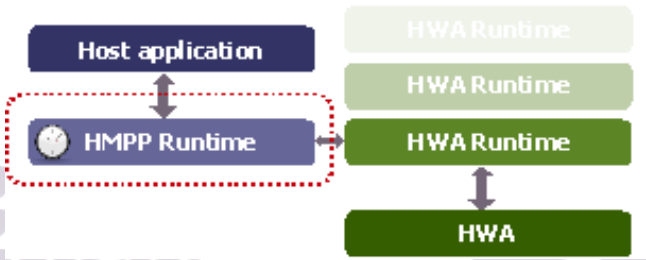
- A set of directives to program hardware accelerators
  - Drive your HWAs, manage transfers



- A complete toolchain to build manycore applications
  - Build your hybrid application



- A runtime to adapt to platform configuration
  - With its API

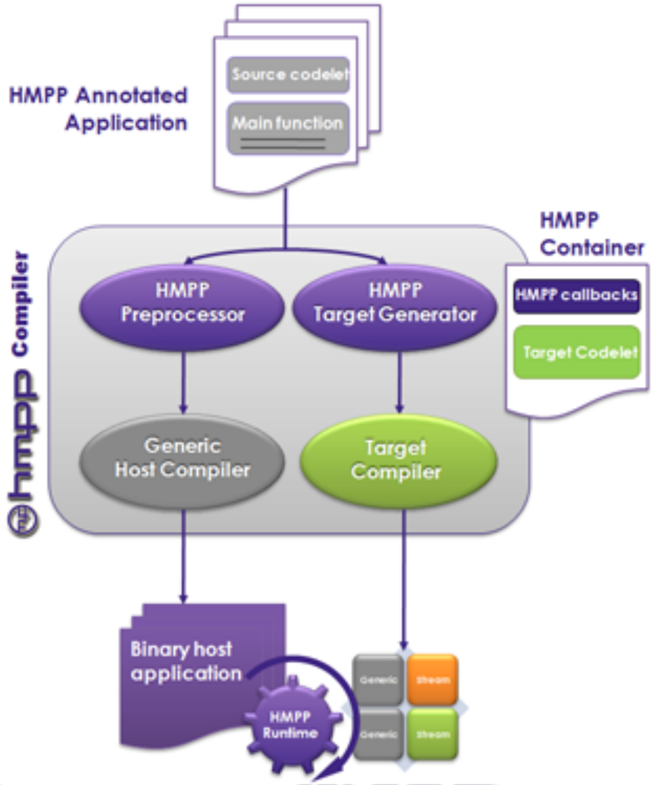


HMPP Overview

- 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
  - Generates CUDA and OpenCL kernels
  - Works with standard compilers and target tools
  - Tuning directives to optimize GPU kernels
- A runtime library
  - Allocates and manages computing resources
  - Dispatches computations on CPU and GPU cores
  - Scales to multi-GPUs systems

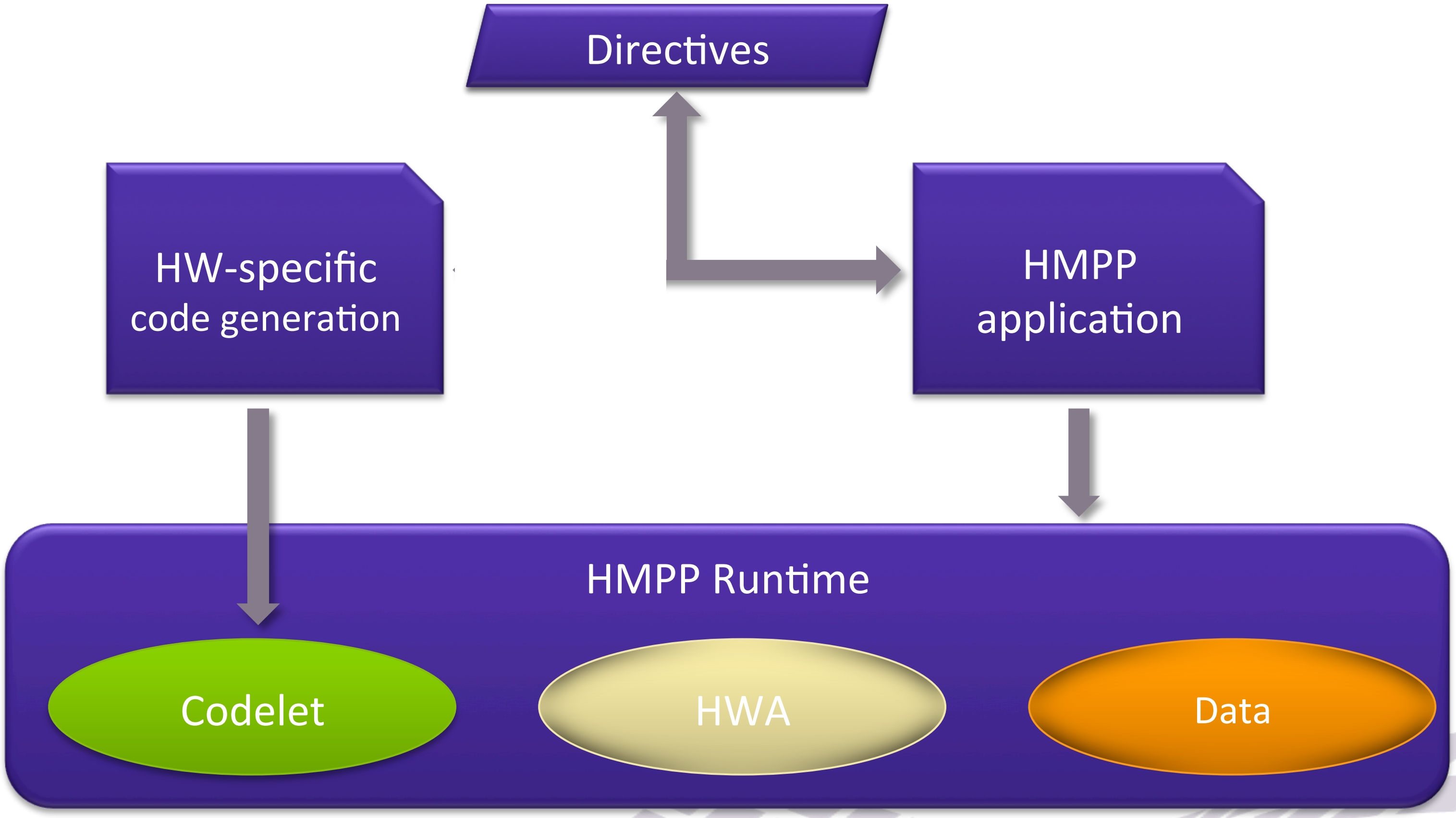
HMPP Compilation Paths

- HMPP drives the whole compilation
  - Host application compilation
    - HMPP runtime is linked to the host part of the application
  - Codelet production
    - Target code is produced
    - A dynamic library is built




```
$ hmpp gcc myProgram.c
```

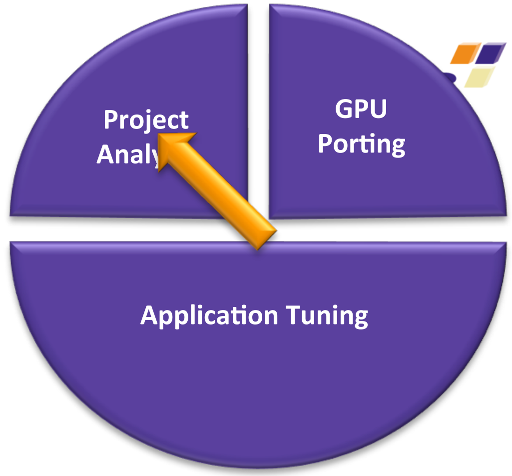
HMPP Directives Drive Hybrid Applications



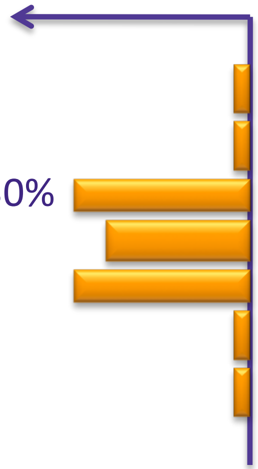
What's New in HMPP 3.0?

- Dynamic data management mechanism
  - Mirrors identified by their host address
  - Simplifies management of data with less directives
- Multi-device programming
  - Exploit multiple devices in one compute node
  - Distribute collections of data over multiple devices
- New run-time API
  - Three bindings for C, C++ and Fortran 90-2003
  - Low level OpenCL style programming with OpenCL/CUDA kernel generation
- Open library integration system
  - CPU and GPU libraries coexist in same binary (proxy mechanism)
  - Data sharing between HMPP user codelets and libraries
  - User can write their own HMPP proxies
  - Proxies provided for cuBLAS, CULA, cuFFT, keeping CPU API.

Step One: Find Hot Spots




```
void derive(int nx, double _Complex …) {
  int i;
  for (i=1; i<nx/2; ++i) {
    wrkq[i] = (0+I-1) * wrkq[i] * cf;
  }
  wrkq[   0] = 0.0+I*0;
  wrkq[nx/2] = 0.0+I*0;
}
```




```
    . . .
    pr2c = fftw_plan_dft_r2c_1d(n, idata_real, …
    pc2r = fftw_plan_dft_c2r_1d(n, odata_intermediate, …
    fftw_execute(pr2c);
    derive(n, odata_intermediate, cf);
    fftw_execute(pc2r);
    fftw_destroy_plan(pr2c);
    fftw_destroy_plan(pc2r);
    . . .
```

- Find hotspots, estimate potential (e.g. Amdahls' Law)
- Check CPU performance, optimize CPU execution
- Setup a validation process
- Estimate parallelism, complexity, …

Analysis of the CPU Code

- Find hotspots, estimate potential (e.g. Amdahls' Law)
  - Using profiling tools
  - gprof, oprofile, …
  - Code instrumentation (gettimeofday(), …)
  - …
- Check CPU performance
  - Is the machine enough loaded ?
  - Optimize CPU execution, CPU code
- Setup a validation process
  - To validate that after each porting steps results are correct
- Estimate hot spots parallelism, complexity, …

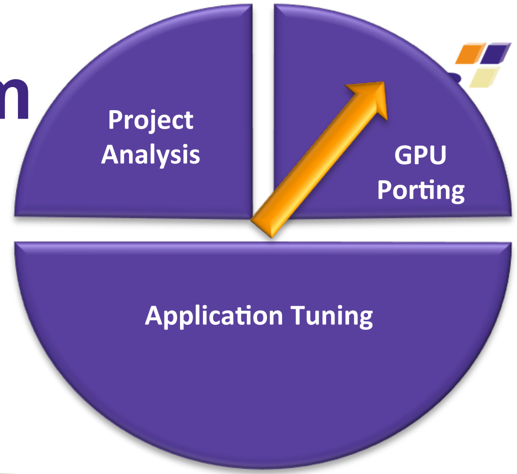
What is a GPU-friendly Profile

- Spikes, bumpy profile
  - Few sections of code to focus on for a good speedup factor
  - The less functions to port, the less cost it involves

- Anyway, a GPU-friendly profile is
  - A profile for which the sections of code to focus on are data-parallel

- Don't forget the Gustafson's law
  - You may discover computational intensive kernels just by varying the amount of their input data
  - Sometimes the parallelism is placed at compute node level, with independent data distributed over the nodes
    - Then gather groups of data onto a same node and parallelize at hardware level

Initial Porting, Highlighting Parallelism



- Exhibit parallelism
- Push the code onto the GPU
- Validate execution

```
#pragma hmpp <g> group, target=CUDA[/OpenCL]
#pragma hmpp <g> derive codelet, args[*]transfer=atcall
```

```
#pragma hmpp <g> group, target=CUDA[/OpenCL]
#pragma hmpp <g> derive codelet, args[*].transfer=atcall
void derive(int nx, double _Complex …) {
  int i;
  for (i=1; i<nx/2; ++i) {
    wrkq[i] = (0+I-1) * wrkq[i] * cf;
  }
  wrkq[   0] = 0.0+I*0;
  wrkq[nx/2] = 0.0+I*0;
}
```

Build a GPU version of the function

Accelerate Codelet Function

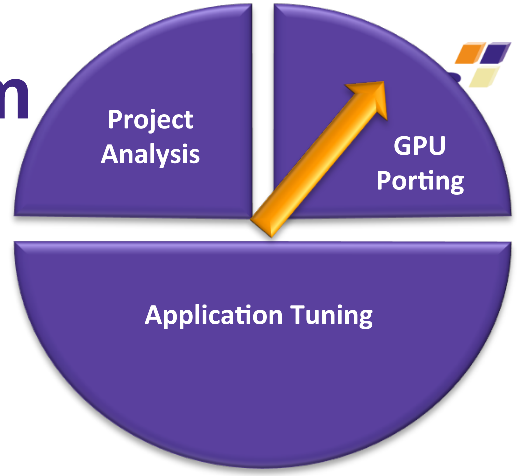
- Declare and call a GPU-accelerated version of a function

```
#pragma hmpp sgemm codelet, target=CUDA:OPENCL, args[*].transfer=atcall
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 < 1000 ; j++ ) {
#pragma hmpp sgemm callsite
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
  }
    /* . . . */
 }
```

Declare CUDA and OPENCL codelets

Synchronous codelet call

Initial Porting, Highlighting Parallelism



- Select implementation for library calls and hotspots
- Insert calls to execute on GPU

Call GPU version of *derive*

```
#pragma hmpp <g> derive callsite
 derive(n, odata_intermediate, cf);
```

```
#pragma hmppalt cufft call, name="fftw_plan_dft_r2c_1d"
    pr2c = fftw_plan_dft_r2c_1d(n, idata_real, …);
#pragma hmppalt cufft call, name="fftw_plan_dft_c2r_1d"
    pc2r = fftw_plan_dft_c2r_1d(n, odata_int
#pragma hmppalt cufft call, name="fftw_execu
    fftw_execute(pr2c);
#pragma hmpp <g> 
    derive(
   . . .
#pragma hmp
    . . .
```

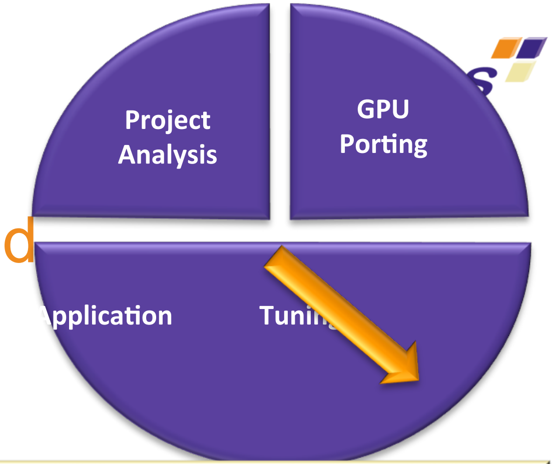
Call GPU version of library call

```
#pragma hmppalt cufft call, name="fftw_execute"
     fftw_execute(pr2c);
```

First Porting Steps using HMPP 3.0

- First thing you want is to validate GPU results
  - If your algorithm produces wrong results
    - Maybe you have a numerical stability problem
    - Or your algorithm is not enough parallel
    - …
- Insert the codelet directive before the definition of the function to offload
  - Use the ATCALL transfer policy
  - HMPP will automatically transfer
    - Scalars as INPUT
    - Arrays, pointers, … as INPUT and OUTPUT
- Insert the HMPPALT directive before calls to library functions
- Validate the result
  - To check that the GPU is a valid target for application
  - It may take time to execute the application
    - Due to all data transfers
    - And not optimized kernels

Transfer Optimizations



- Reduce CPU-GPU communication overhead
- Exploit reuse of data on the GPU

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

 for( j = 0 ; j < 1000 ; j++ ) {
#pragma hmpp sgemm
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );

 }
 . . .
#pragma hmpp sgemm delegatedstore, data[vout]

#pragma hmpp sgemm free
#pragma hmpp sgemm release
```

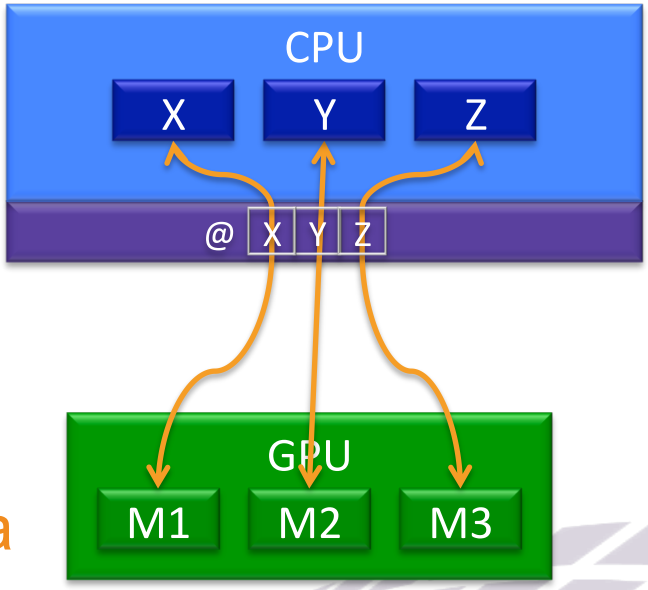
Preload data

Iterate 1000 times without data transfer

Download results

Storage Policy

- Mirrored data or simply mirror
  - An area of memory on the host is mirrored on the accelerator
  - The HMPP runtime dynamically makes the link between the host address and the device address



- Simple data management
  - Few directives to manage mirrored data
- Easy to dynamically allocate and free a mirror
  - Use the ALLOCATE and FREE directives

Compute Asynchronously

- Perform CPU/GPU computations asynchronously

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

 for( j = 0 ; j < 1000 ; j++ ) {
#pragma hmpp sgemm callsite, asynchronous
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
    /* . . . */
  }

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

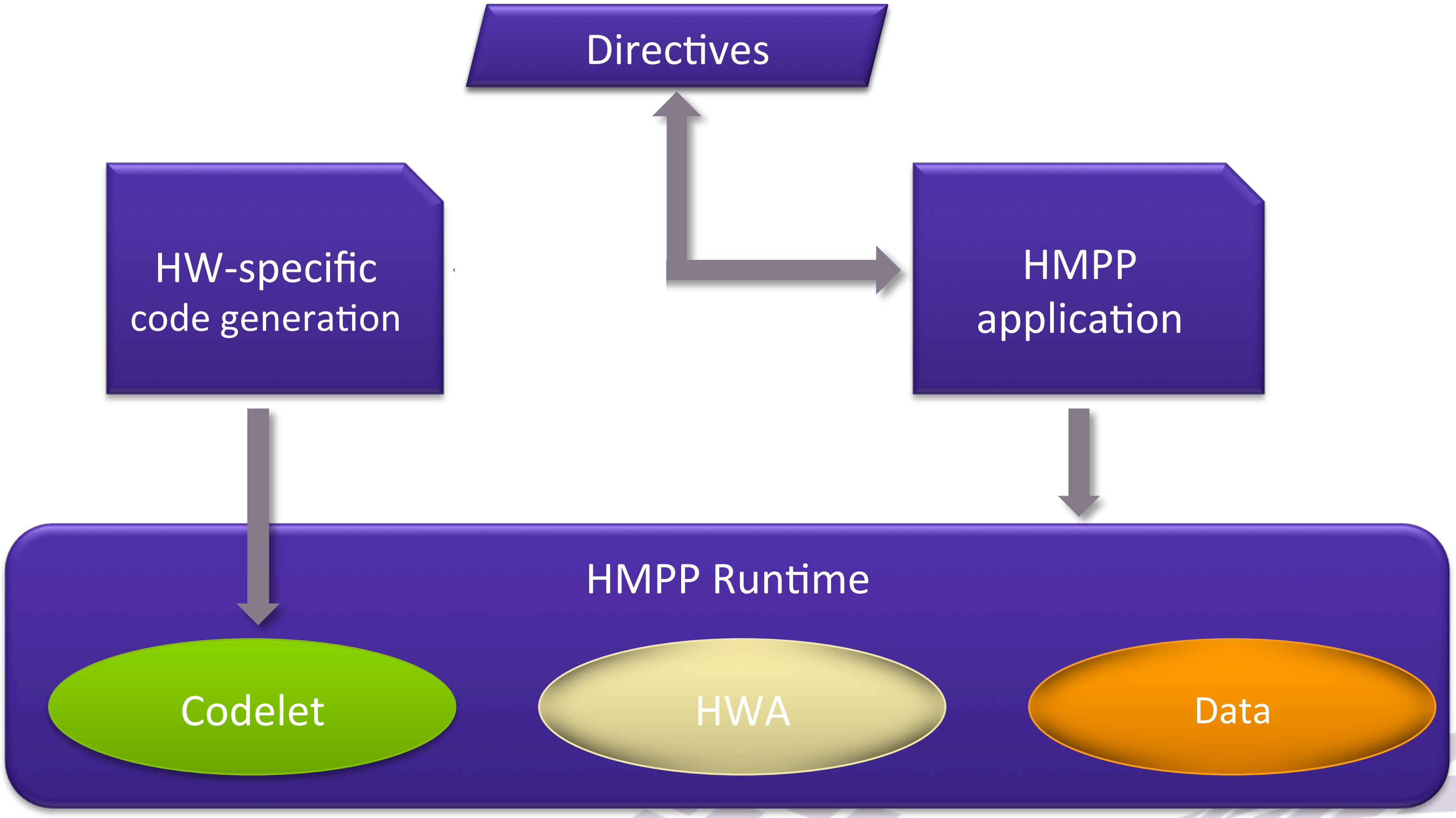
Execute asynchronously

HMPP Directives Overview

- CODELET : Specialize a subroutine
- CALLSITE : Specialize a call statement
- SYNCHRONIZE : Wait for completion of the callsite
- ACQUIRE : Set a device for the execution
- ALLOCATE : Allocate memory
- FREE : Free allocated memory
- RELEASE : Release HWA
- ADVANCEDLOAD : Explicit data transfer CPU -> HWA
- DELEGATEDSTORE : Explicit data transfer HWA -> CPU
- GROUP : Groups codelets

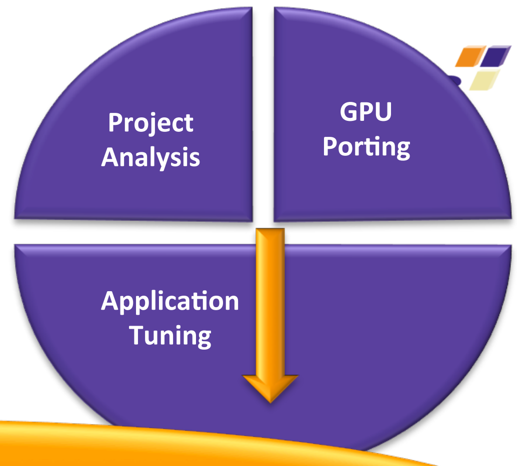
» Directives in green are declarative
» Directives in Red are operational

What About Directives for Code Generation?



Improving Code Generation

- Directive-based GPU kernel code transformations



#pragma hmppcg unroll(4),split, noremainder

```
#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];
   }
 }
```

Use pragma to preserve CPU code

Codelet Tuning Directives for High Level Optimization

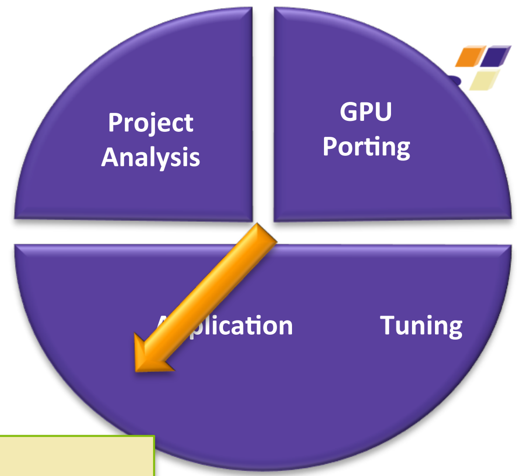
CAPS

hmpp

High level application tweaking

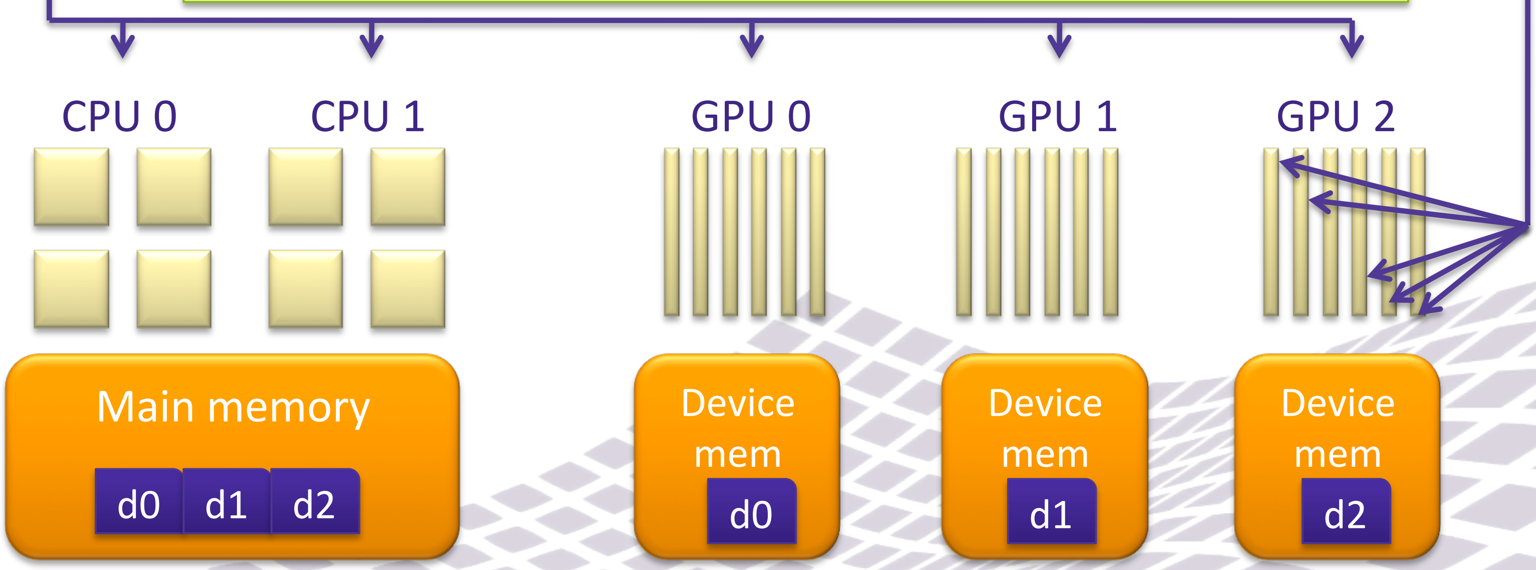
- By adding properties
  - 1D or 2D gridification
- Applying code transformations
  - Loop tiling, unroll, jam, permute, fuse, …
- Using target specific directives
  - Micro architecture management (warp size…)
  - Memory management (CUDA shared memory, constant…)

Scaling to Many-many cores



- Spread computations on available devices
- Manage data over several memory spaces

```
float data[n][x][y];
#pragma hmpp parallel, device="k%3"
for(k=0;k<n;k++) {
     #pragma hmpp <MyGroup> f1 callsite
         myparallelfunc(&data[k],n);
  }
```



Multi-GPUs sample

```
#pragma hmpp <mygroup> group, target=CUDA

#pragma hmpp <mygroup> doit codelet, args[*].mirror,     &
#pragma hmpp    &        args[*].transfer=manual
void doit(float A[1234]) {
...
}

float X[100][1234] ;  // I have 100 arrays
#pragma hmpp <mygroup> acquire, device=0
#pragma hmpp <mygroup> acquire, device=1
...
for (k=0;k<100;k++) {
  float *ptr = X[i] ;
  #pragma hmpp <mygroup> allocate, data[ptr], size={1234}, &
  #pragma hmpp                     & device="k%2"
}
#pragma hmpp parallel
for (k=0;k<100;k++) {
  #pragma hmpp <mygroup> advancedload, data["X[k]"]
  #pragma hmpp <mygroup> doit callsite
  doit(X[k]) ;
  #pragma hmpp <mygroup> delegatedstore, data["X[k]"]
}
```

Acquire two devices

Allocate data on a device then the other

Execute the codelets on the device that owns each mirror

Extern Functions

- Support for function calls inside codelets or regions
  - Functions called in codelets can be defined in other files
  - Avoid code duplication

sum.h

```
#ifndef SUM_H
#define SUM_H

float sum( float x, float y );

#endif /* SUM_H */
```

sum.c

```
#include "sum.h"

#pragma hmpp function,target=CUDA
float sum( float x, float y ) {
  return x+y;
}
```

extern.c

```
#include "sum.h"

int main(int argc, char **argv) {
  int i, N = 64;
  float A[N], B[N];
  ...
  #pragma hmpp cdlt region, args[B].io=inout, target=CUDA
  {
    #pragma hmppcg extern, sum

    for( int i = 0 ; i < N ; i++ )
      B[i] = sum( A[i], B[i] );
  }
  ...
}
```

Import external declaration

'sum' is an external function

Declare 'sum' as a function called in a codelet

HMPP Runtime API

- Available bindings in C/C++ and Fortran
  - Low level OpenCL style programming with OpenCL/CUDA kernel generation
  - C++ API throws exceptions
- API call allows you to
  - Acquire a device
  - Allocate data
  - Transfer data
  - Launch codelets
  - Free data
  - Asynchronous operations
  - …
- Really useful for C++ programmers

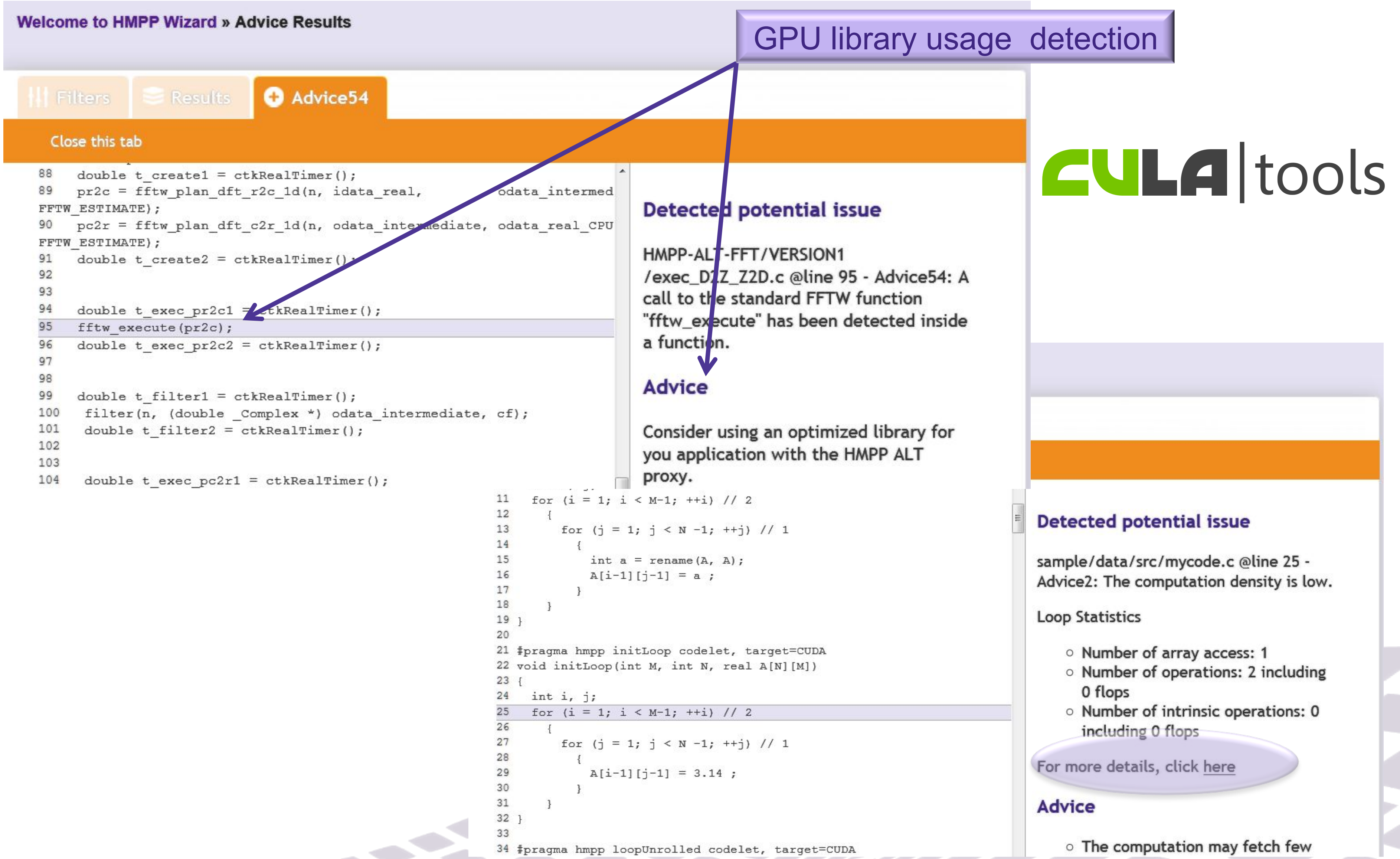
HMPP3 Summary

- Abstract the programming of manycore architectures
  - A rich set of programming and tuning directives
  - Distribute computations to exploit CPU and GPU cores in a node
  - Mix CPU and GPU libraries in same binary
  - Incrementally develop and port applications

- An open source-to-source compiler
  - Work with standard compilers and hardware vendor tools
  - Ease maintenance by avoiding different languages
  - Preserve legacy code

HMPP Wizard

HMPP Wizard

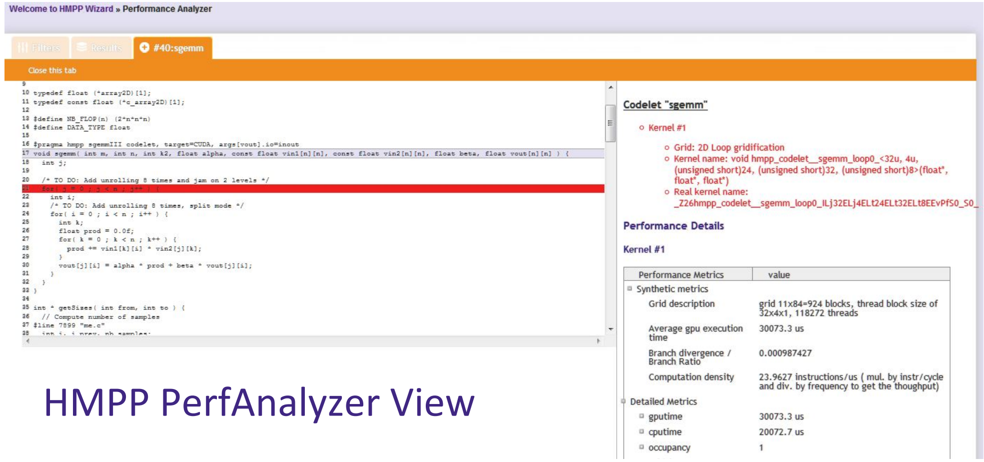

Welcome to HMPP Wizard » Advice Results
GPU library usage detection
Filters
Results
Advice54
Close this tab
double t_create1 = ctkRealTimer();
pr2c = fftw_plan_dft_r2c_1d(n, idata_real, odata_intermed
FFTW_ESTIMATE);
pc2r = fftw_plan_dft_c2r_1d(n, odata_intermediate, odata_real_CPU
FFTW_ESTIMATE);
double t_create2 = ctkRealTimer();
double t_exec_pr2c1 = ctkRealTimer();
fftw_execute(pr2c);
double t_exec_pr2c2 = ctkRealTimer();
double t_filter1 = ctkRealTimer();
filter(n, (double _Complex *) odata_intermediate, cf);
double t_filter2 = ctkRealTimer();
double t_exec_pc2r1 = ctkRealTimer();
Detected potential issue
HMPP-ALT-FFT/VERSION1 /exec_D2Z_Z2D.c @line 95 - Advice54: A call to the standard FFTW function "fftw_execute" has been detected inside a function.
Advice
Consider using an optimized library for you application with the HMPP ALT proxy.
for (i = 1; i < M-1; ++i) // 2
{
for (j = 1; j < N -1; ++j) // 1
{
int a = rename(A, A);
A[i-1][j-1] = a ;
}
}
}
#pragma hmpp initLoop codelet, target=CUDA
void initLoop(int M, int N, real A[N][M])
{
int i, j;
for (i = 1; i < M-1; ++i) // 2
{
for (j = 1; j < N -1; ++j) // 1
{
A[i-1][j-1] = 3.14 ;
}
}
}
#pragma hmpp loopUnrolled codelet, target=CUDA
Detected potential issue
sample/data/src/mycode.c @line 25 - Advice2: The computation density is low.
Loop Statistics
Number of array access: 1
Number of operations: 2 including 0 flops
Number of intrinsic operations: 0 including 0 flops
For more details, click here
Advice
The computation may fetch few

HMPP Wizard

- HMPP wizard synthetizes metrics based on static and dynamic information
  - The result is shown as a HTML page
- Getting dynamic information from profilers
  - Gprof
  - Oprofile
- Getting static information from code analysis
  - Library calls
  - Code transformation inside codelets

Performance Measurements

- Parallelism is about performance!
- Track Amdahl's Law Issues
  - Serial execution is a killer
  - Check scalability
  - Use performance tools



HMPP PerfAnalyzer View

- Add performance measurement in the code
  - Detect bottleneck asap
  - Make it part of the validation process

Accelerator Programming Model

Parallelization

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

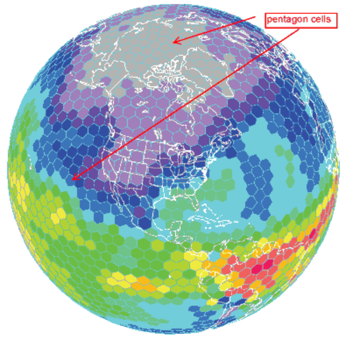
http://www.caps-entreprise.com

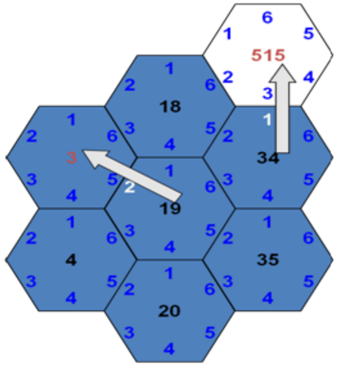
http://twitter.com/CAPSentreprise

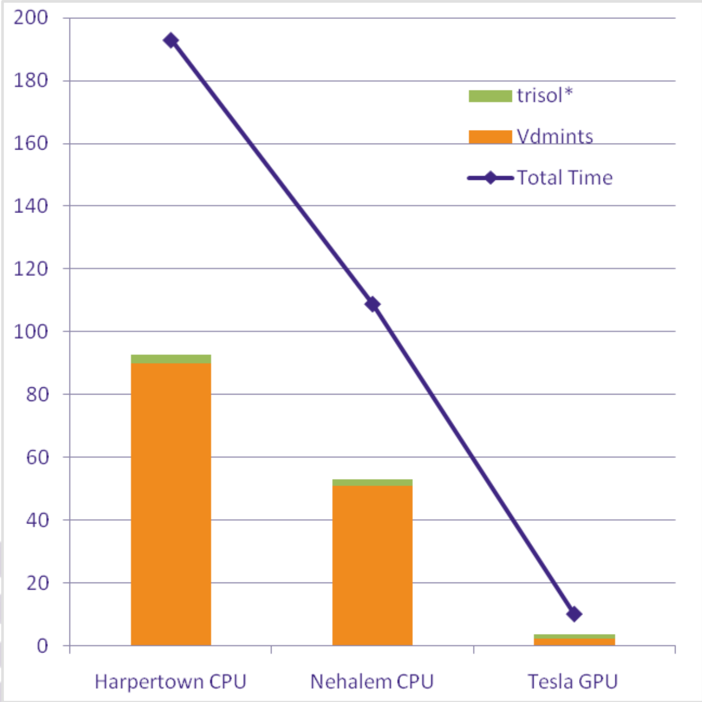
Weather Forecasting

A global cloud resolving model

- Resource spent
  - 1 man-month (part of the code already ported)
- GPU C1060 improvement
  - 11x over serial code on Nehalem
- Main porting operation
  - reduction of CPU-GPU transfers
- Main difficulty
  - GPU memory size is the limiting factor



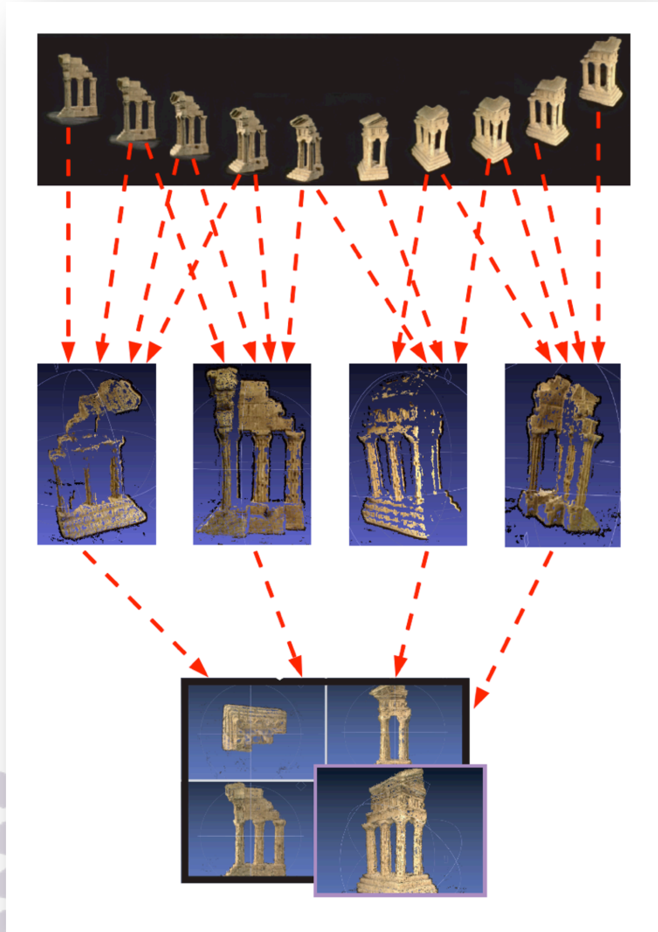




Computer vision & Medical imaging

MultiView Stereo

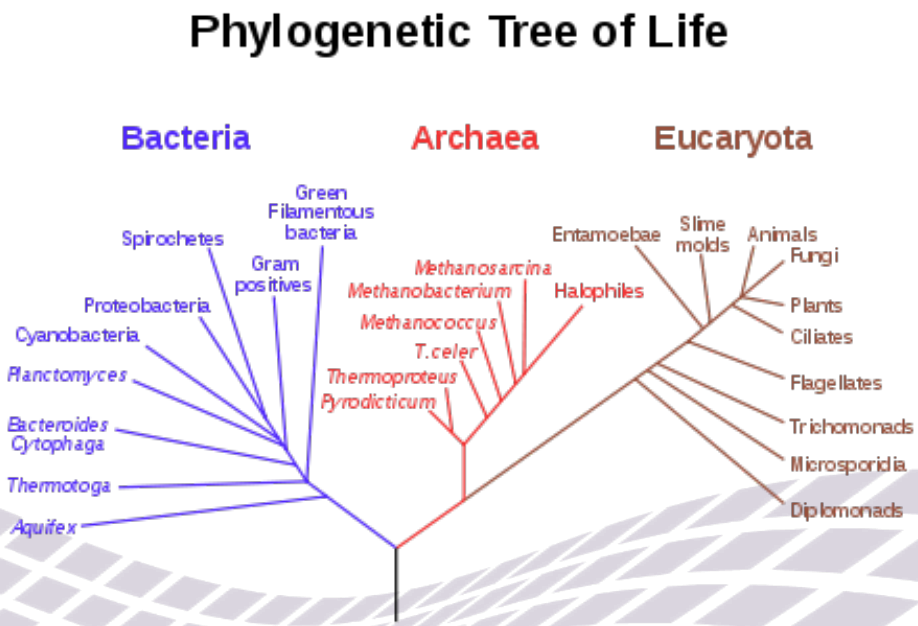
- Resource spent
  - 1 man-month
- Size
  - ~1kLoC of C99 (DP)
- CPU Improvement
  - x 4,86
- GPU C2050 improvement
  - x 120 over serial code on Nehalem
- Main porting operation
  - Rethinking algorithm



Biosciences, phylogenetics

Phylip, DNA distance

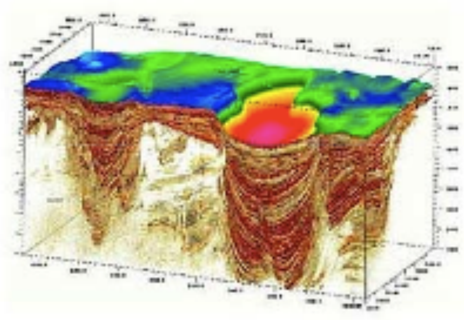
- In association with the HMPP Center Of Excellence for APAC
- Computes a matric of distances between DNA distances
- Resource spent
    - A first CUDA version developed by Shanghai Jiao Tong University, HPC Lab
    - 1 man-month
- Size
    - 8700 lines of C code, one main kernel (99% of the execution time)
- GPU C2070 improvement
    - x 44 over serial code on Nehalem
- Main porting operation
    - Kernel parallelism & data transfer coalescing leverage
    - Conversion from double precision to simple precision computation

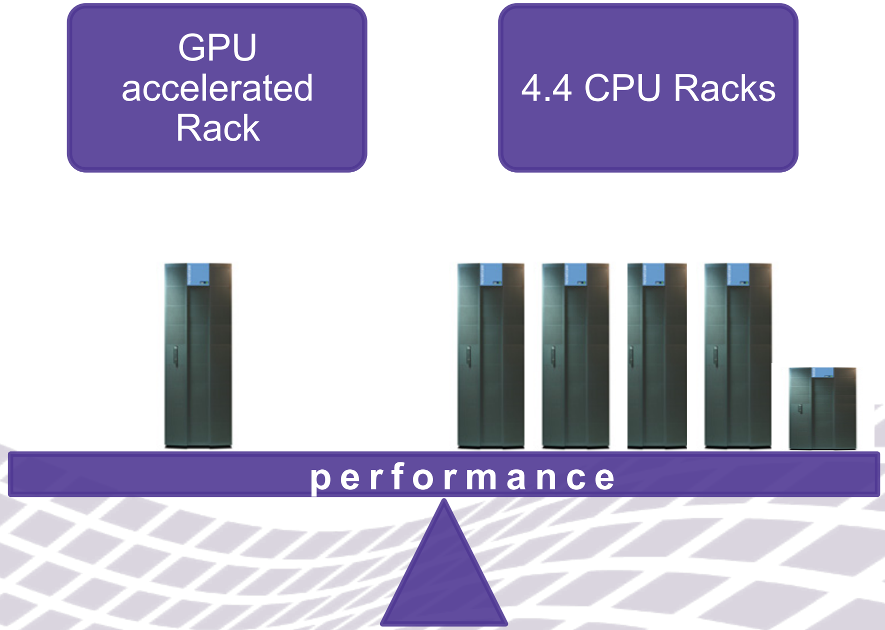


Oil & Gas

GPU-accelerated seismic depth imaging

- 1 GPU accelerated machine = 4.4 CPU machines
  - GPU: 16 dual socket quadcore Intel Hapertown nodes
    connected to 32 GPUs
  - CPU: 64 dual socket quadcore Intel Hapertown nodes





energie atomique • energies alternatives

Performances (max)
NO TRANSFERTS

T in s

Scalar	18457.52	3030.9	4668.67	7531.1	1054.4	15.68	1933.82
OMP=8	5040.1	379.09	1479.91	1302.64	1248.35	6.76	348.32
HMPP	820.34	56.88	388.23	156.61	81.58	2.04	29.7
CUDA	1267.66	75.01	458.95	135.61	66.99	66.51	531.3
	ALL	NOISE	DIFFUS	KERSBS	SHIFT	BOUND	FLUX
OMP / SEQ	3.66	8.00	3.15	5.78	0.84	2.32	5.55
HMPP / SEQ	22.50	53.29	12.03	48.09	12.92	7.69	65.11
CUDA / SEQ	14.56	40.41	10.17	55.53	15.74	0.24	3.64

Speedup

Geom: 128 x 128 x 256
No I/O
1 MPI

Diffus = FFT FW + diffrac + FFTBW
KERSBS = KER + SBS

GCdV CEA, DAM, DIF, F-91297 Arpajon 2

Guillaume Colin de Verdière, Onera XtremCFD Workshop, 7th of October, 2011