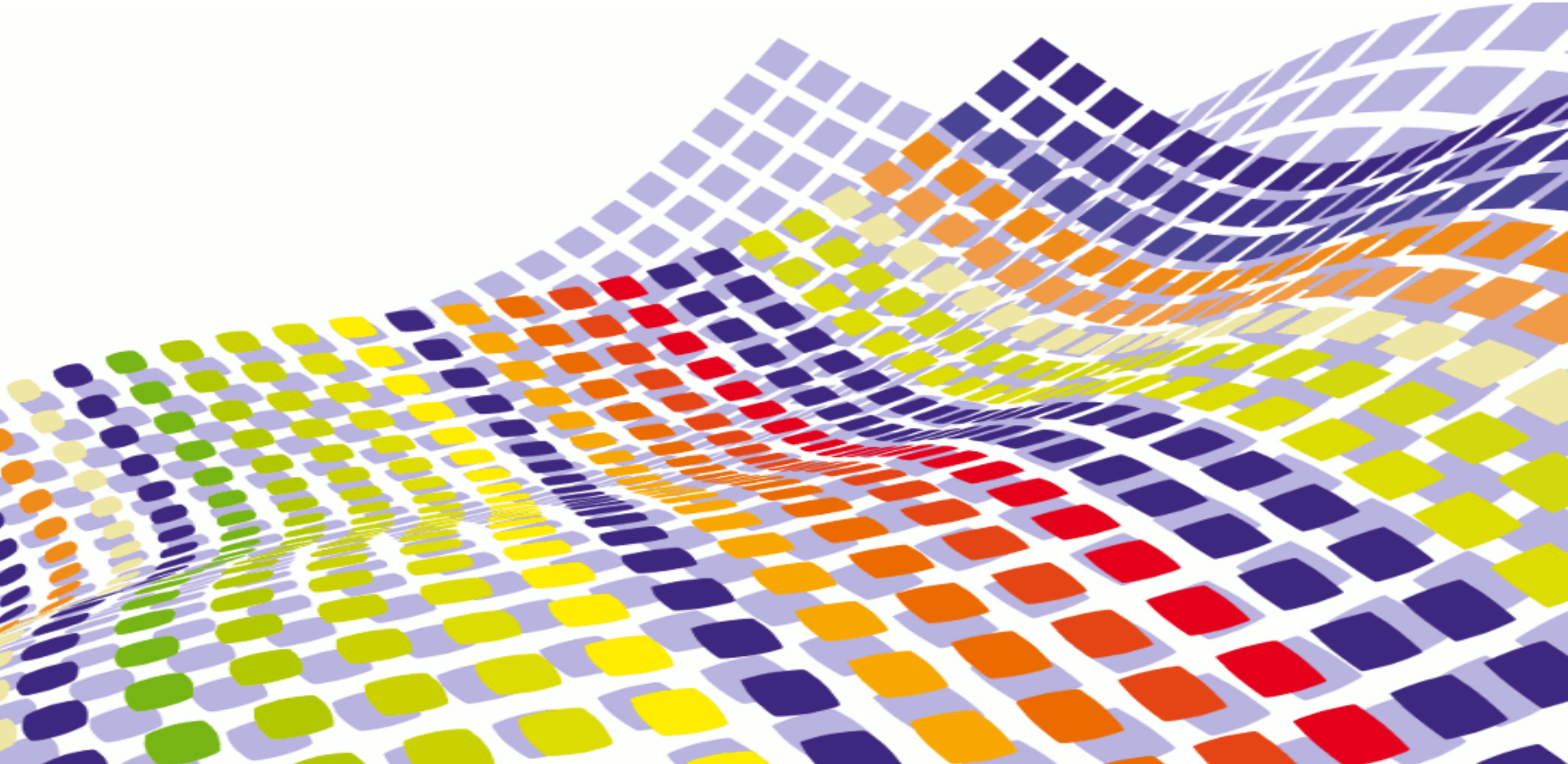
CAPS

HMPP Workbench

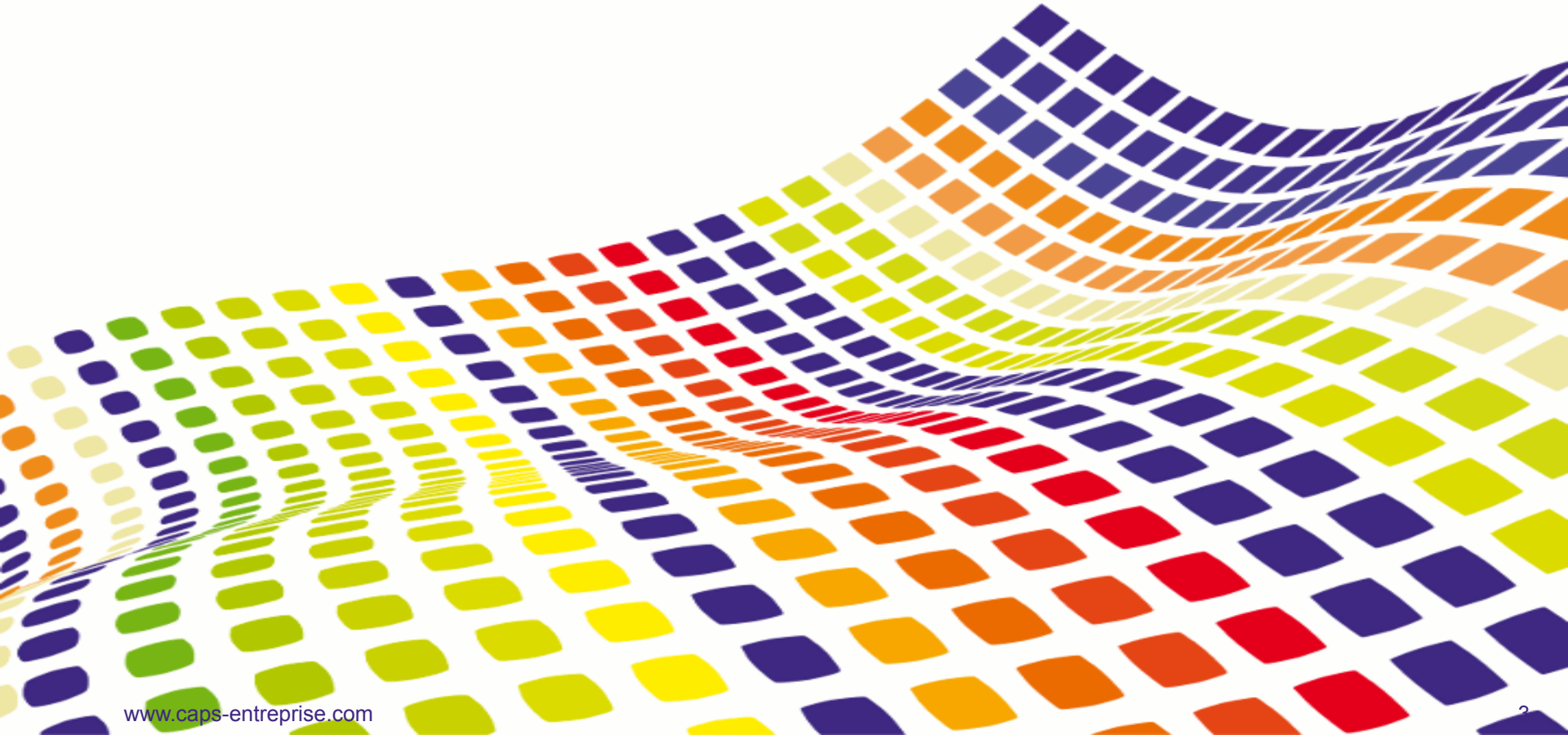
Technical Overview



Agenda

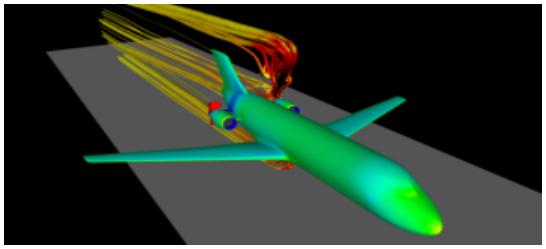
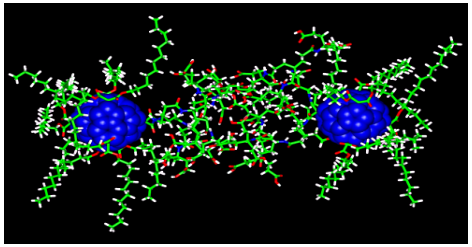
- GPU & Hybrid Parallel computing
- HMPP Overview
- HMPP Features & Roadmap

GPGPU & Parallel Hybrid Computing

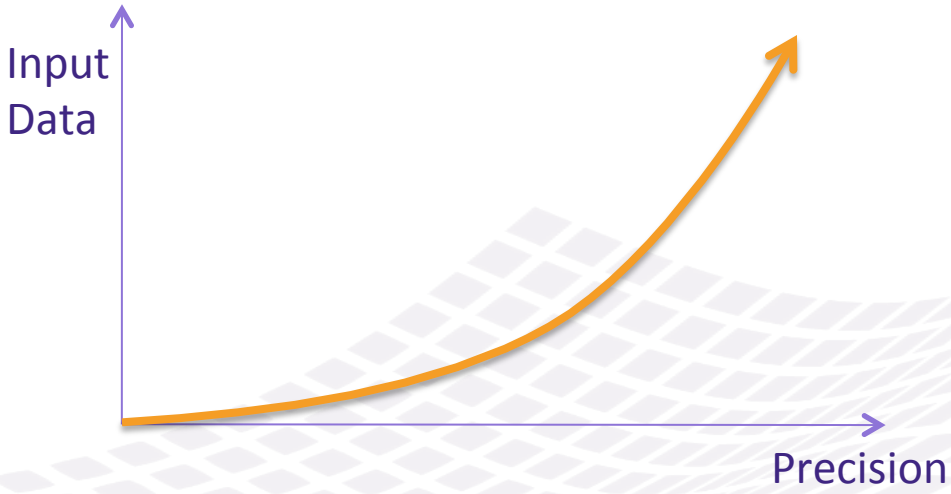


Industry and Business Facts

- Modeling & Simulation are pervasive

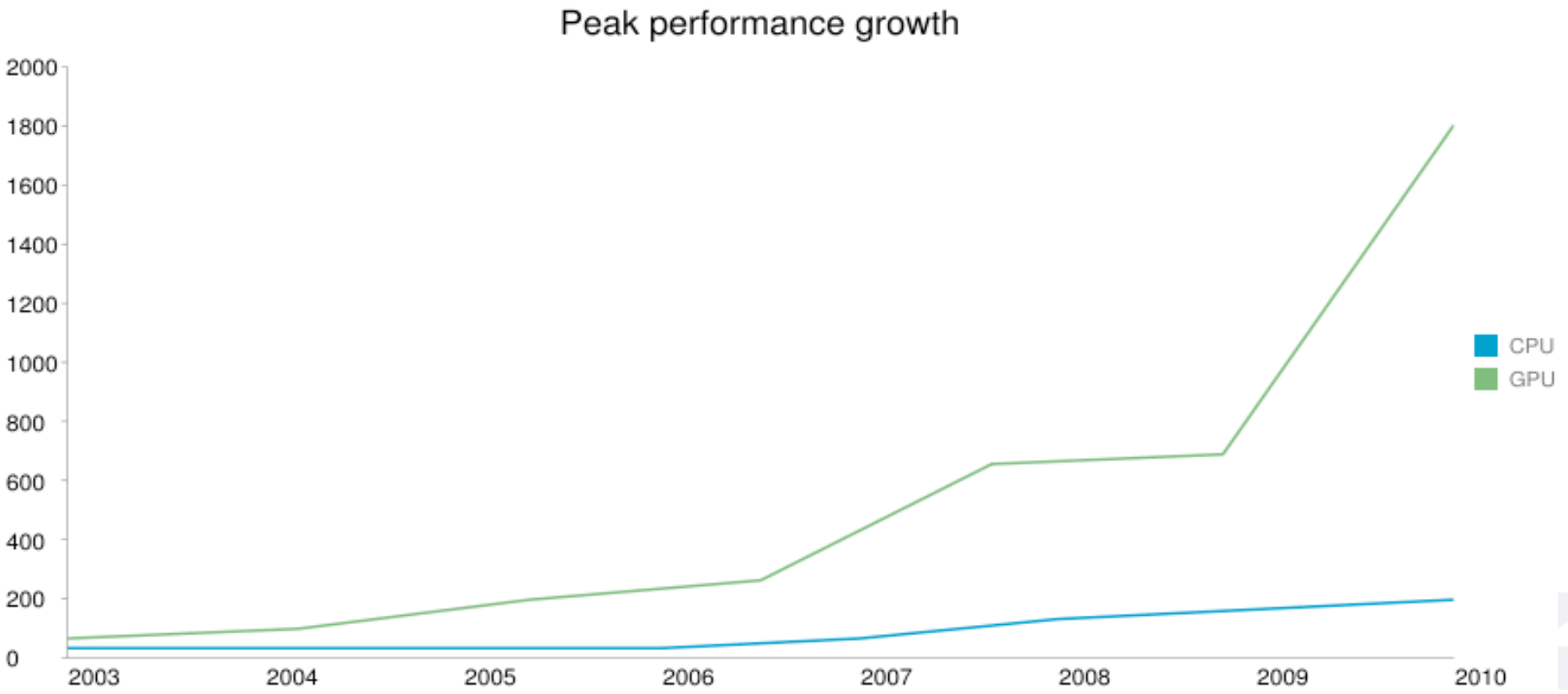

- Precision is the key to success



Why Hybrid Computing?

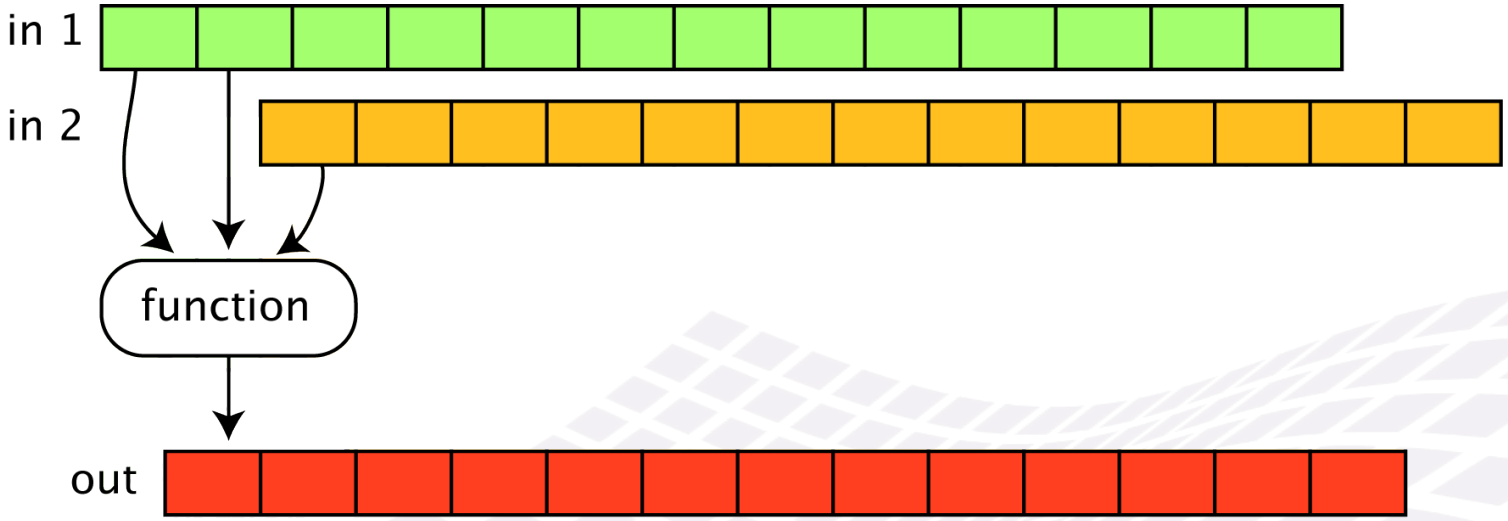
- You need more precision / more speed in your computations
- Current technologies reached their limits (frequency)
- One solution, use parallelism (increase # of cores)
  - we do more things in // instead of doing it quicker
  - Operation/Watt is the efficiency scale
- Mainstream applications will rely on these multicore / manycore architectures
- Various heterogeneous hardware
  - General purpose cores
  - Application specific cores (e.g. GPUs)

Parallel Processor Architectures

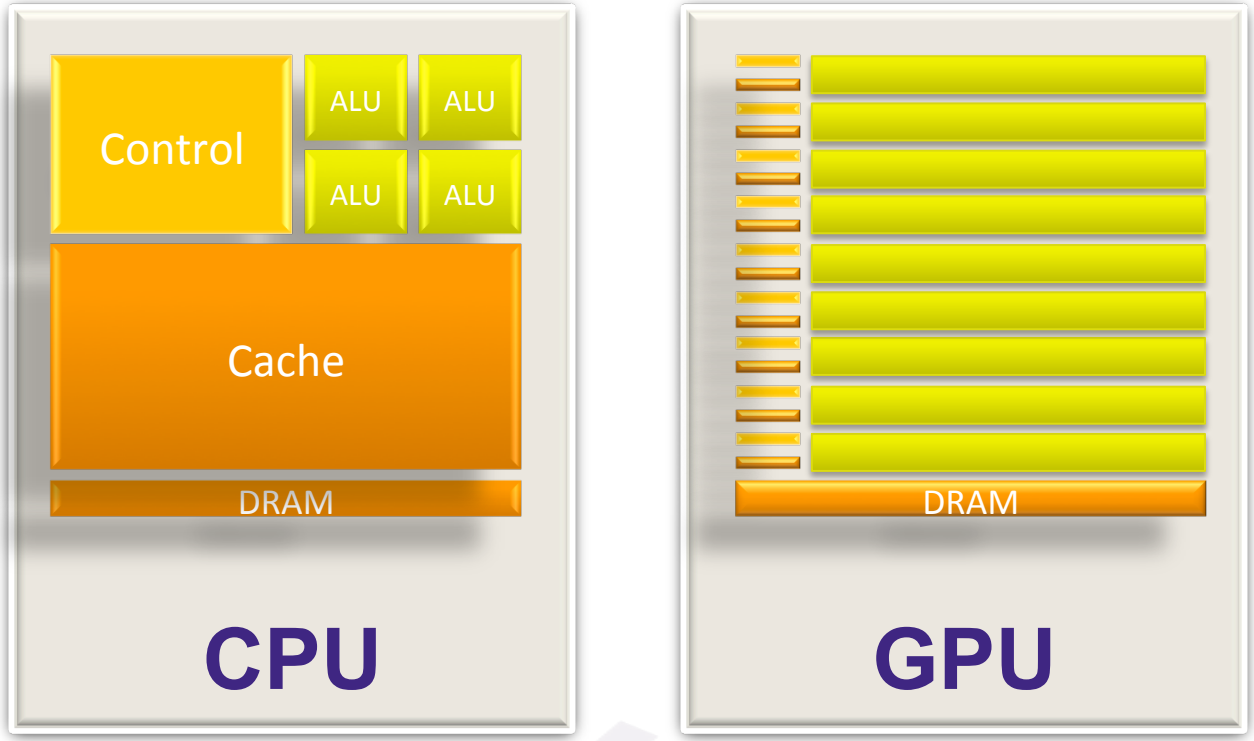


GPU computing

- Stream programming is well suited to GPU
  - But memory hierarchy is exposed
- A similar computation is performed on a collection of data (*stream*)
  - There is no data dependence between the computation on different stream elements



Parallel Processor Architectures



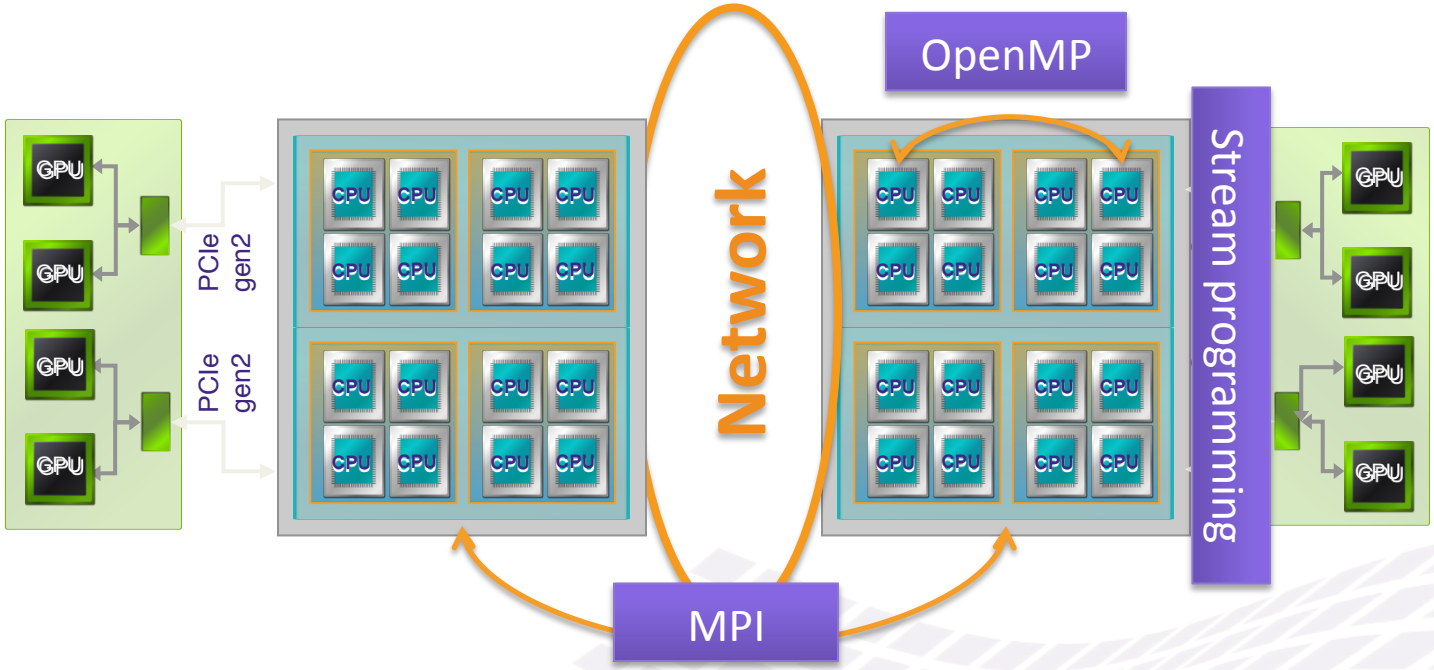
General purpose architecture

Massively data parallel

Needs 1000s of computation threads to be efficient

Multiple Parallelism Levels

- Programming various hardware components of a node cannot be done separately



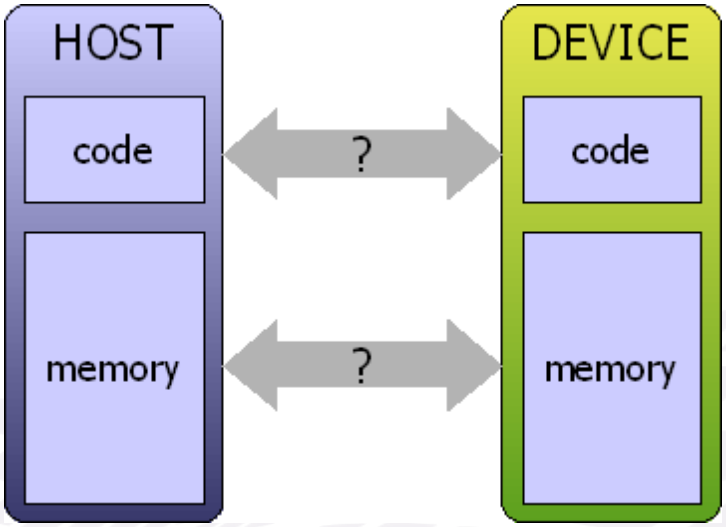
Offloading computations

- Host: General purpose cores
  - Share a main memory
  - Core ISA provides fast SIMD instructions
- Device: Streaming cores
  - GPU, DSP, FPGA… (vector, SIMD)
  - Application specific architectures ("*narrowband*")
  - Can be extremely fast
- Hundreds of GigaOps
  - But not easy to leverage
  - Restriction to one platform is not acceptable

Key Issues

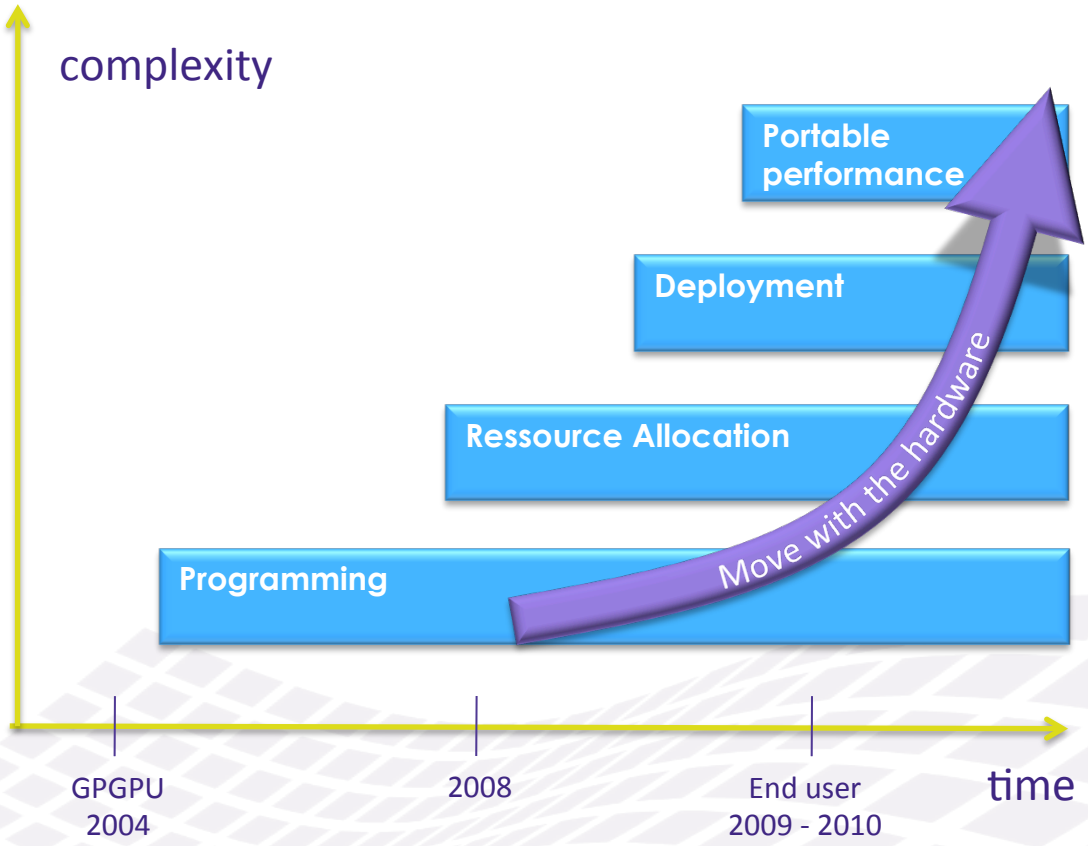
- Software should stay hardware-independent
  - New architectures / languages to master
  - Hybrid solutions evolve -> redo the work each time it changes

- Huge potential performance but accelerators are far from host memory
  - Data must be copied on the remote device
  - Due to narrowband links between CPU/HWA, data transfers are critical

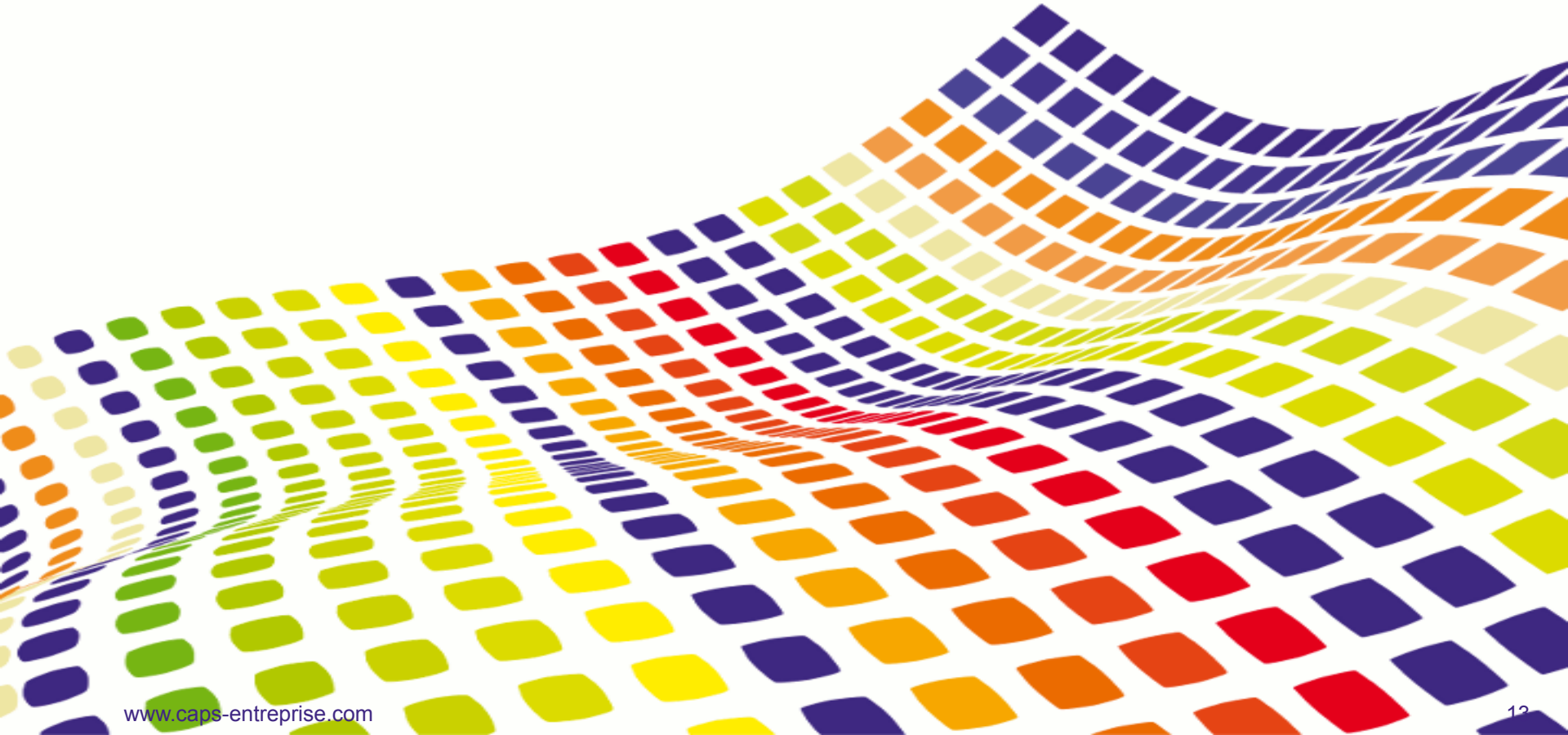


Manycore Challenges

- Programming
  - Medium
- Resources management
  - Medium
- Application deployment
  - Hard
- Portable performance
  - Extremely hard

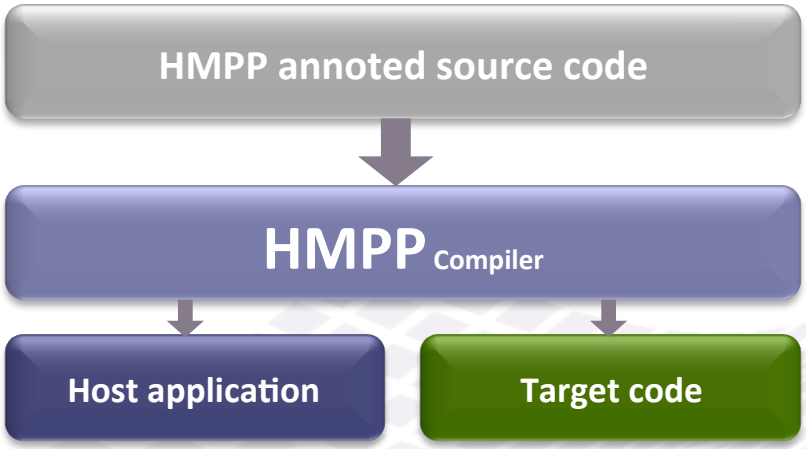


HMPP Overview



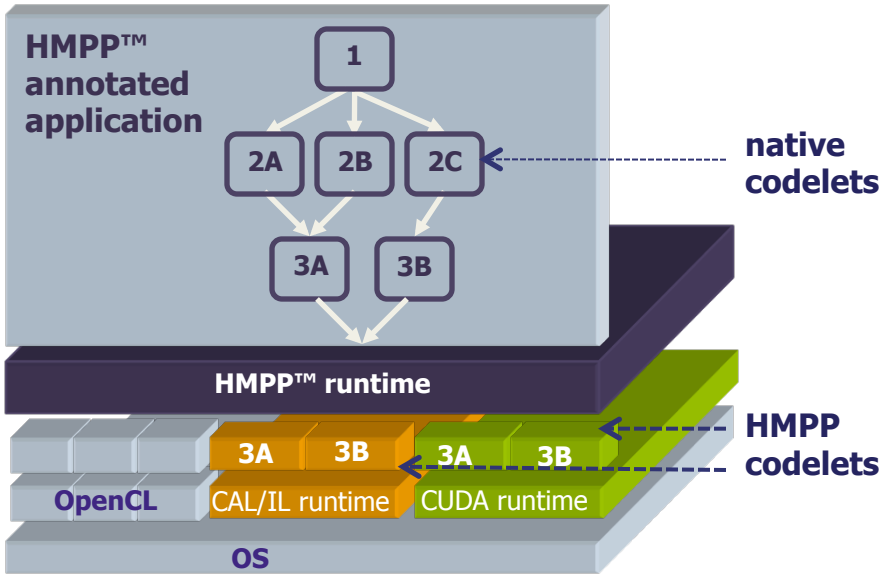
High Level Abstraction of Stream Programming

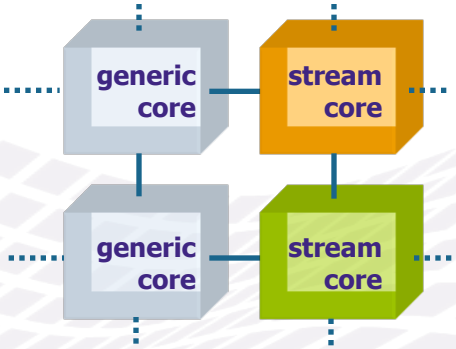
- C & Fortran programming directives
  - Preserve legacy code
- A compiler integrating GPU stream code generators
  - Insulate hardware specific implementations
- Runtime library
  - Ease application deployment



Application Execution

- HMPP detects presence of available GPUs
- Fall back to native version when fail





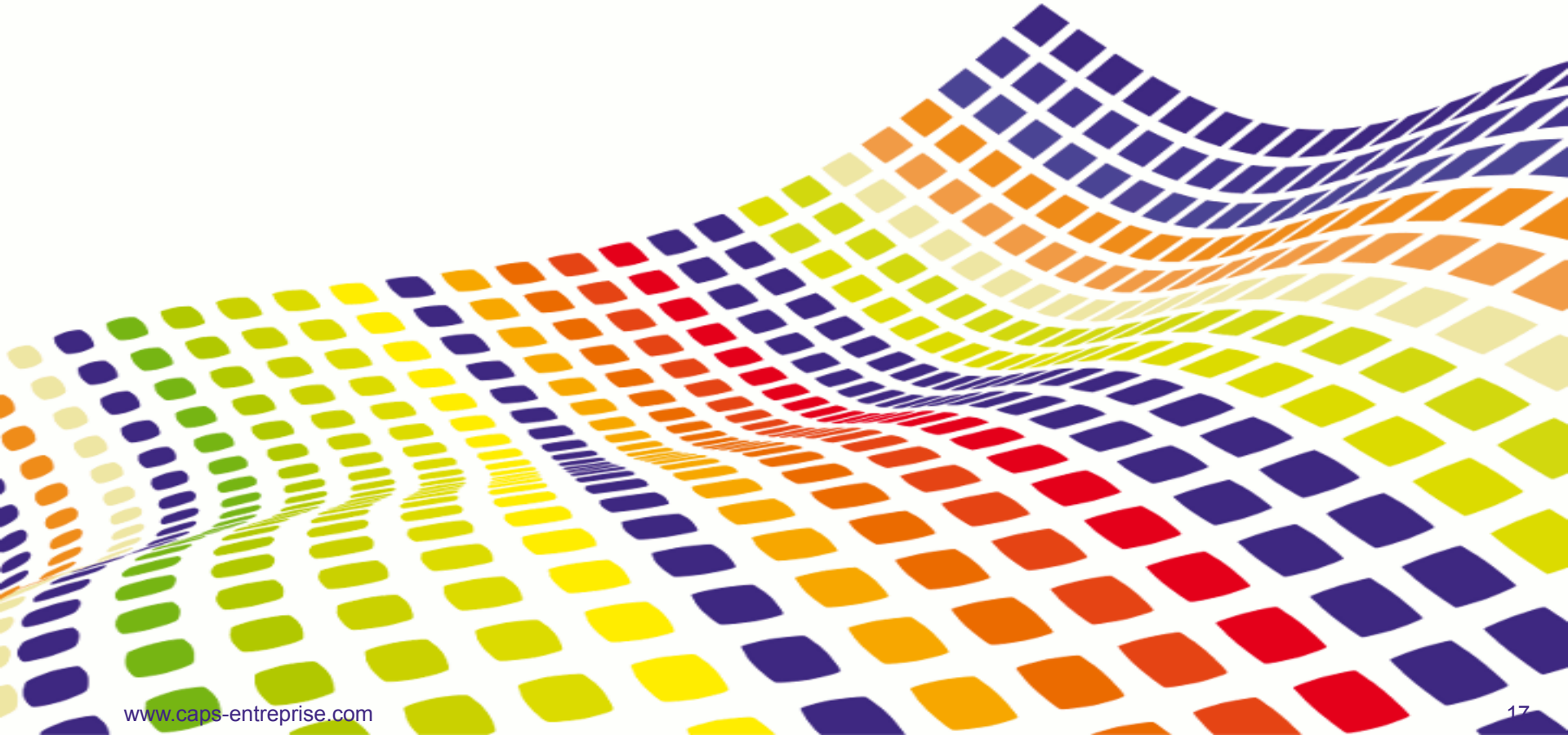
HMPP Mission

CAPS

**Think manycore once,
deploy on multiple accelerators**

- Efficiently orchestrate parallel CPU/GPU computations in legacy code
- Automatically produce powerful manycore applications
- Ease application development and deployment
- Preserve software assets

Preserve Software Assets



HMPP Directives

- Preserve legacy code
- Complementary with other programming tools
- No exit cost

Directives is the best way for manycore programming

HMPP C

A single line directive is:

```
#pragma hmpp label command [ , attribute … ]
```

HMPP Fortran

A single line directive is:

```
!$hmpp label command [ ,  attribute … ]
```

Preserve software assets

Hybrid Computing Power

- Two directives to access hybrid computing power

```
#pragma hmpp sgemm 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 sgemm callsite
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
  }
 …
```

CODELET : Specialize a subroutine

CALLSITE : Specialize a call statement

Preserve software assets

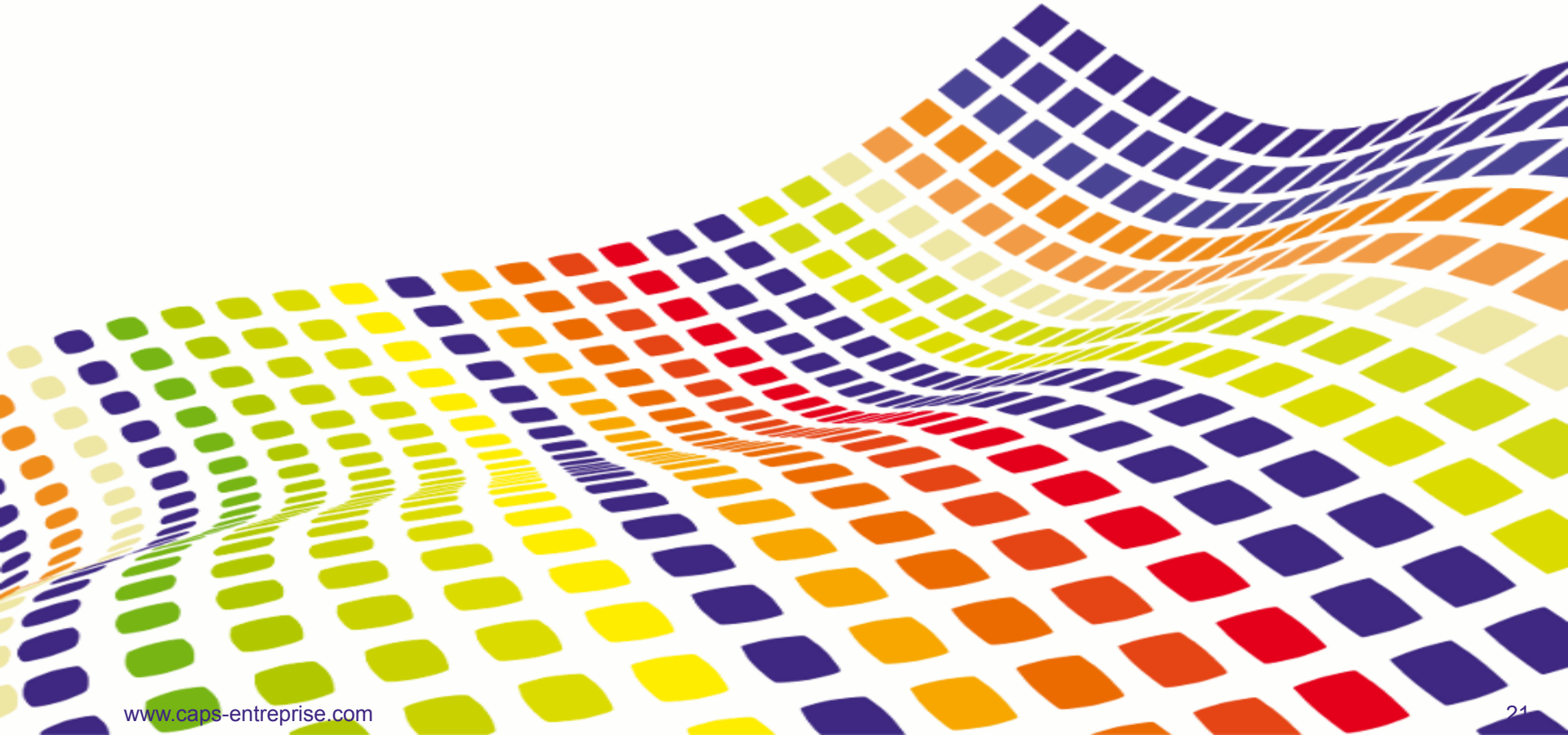
Debugging HMPP Applications

- HMPP applications are still compatible with usual debuggers
  - Compile with “-g” and debug with Gdb
  - Compile with “-G” and debug with Cuda-dbg
  - …

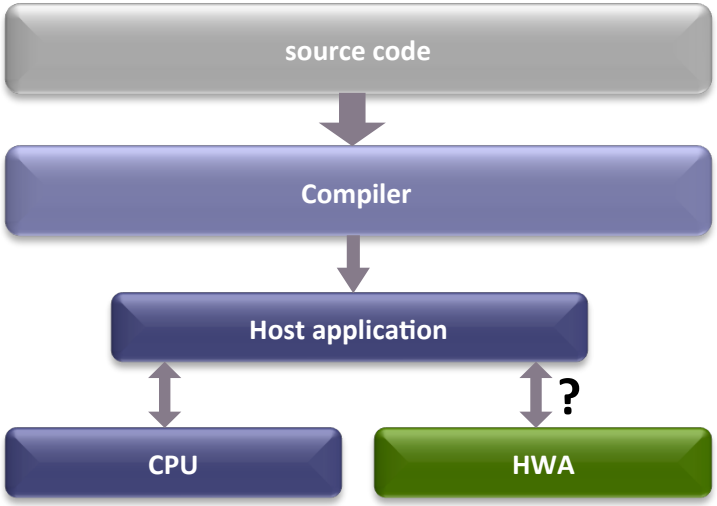
- We are working on the integration of HMPP to parallel debuggers

Preserve software assets

Ease Application Deployment



Hybrid Software for Hybrid Hardware



- Some decisions have to be taken at runtime
  - An hybrid application must be aware of its environment to get the best of hardware resources
    - Is there any accelerator available? What kind is it?
    - Can this application run without any accelerator?

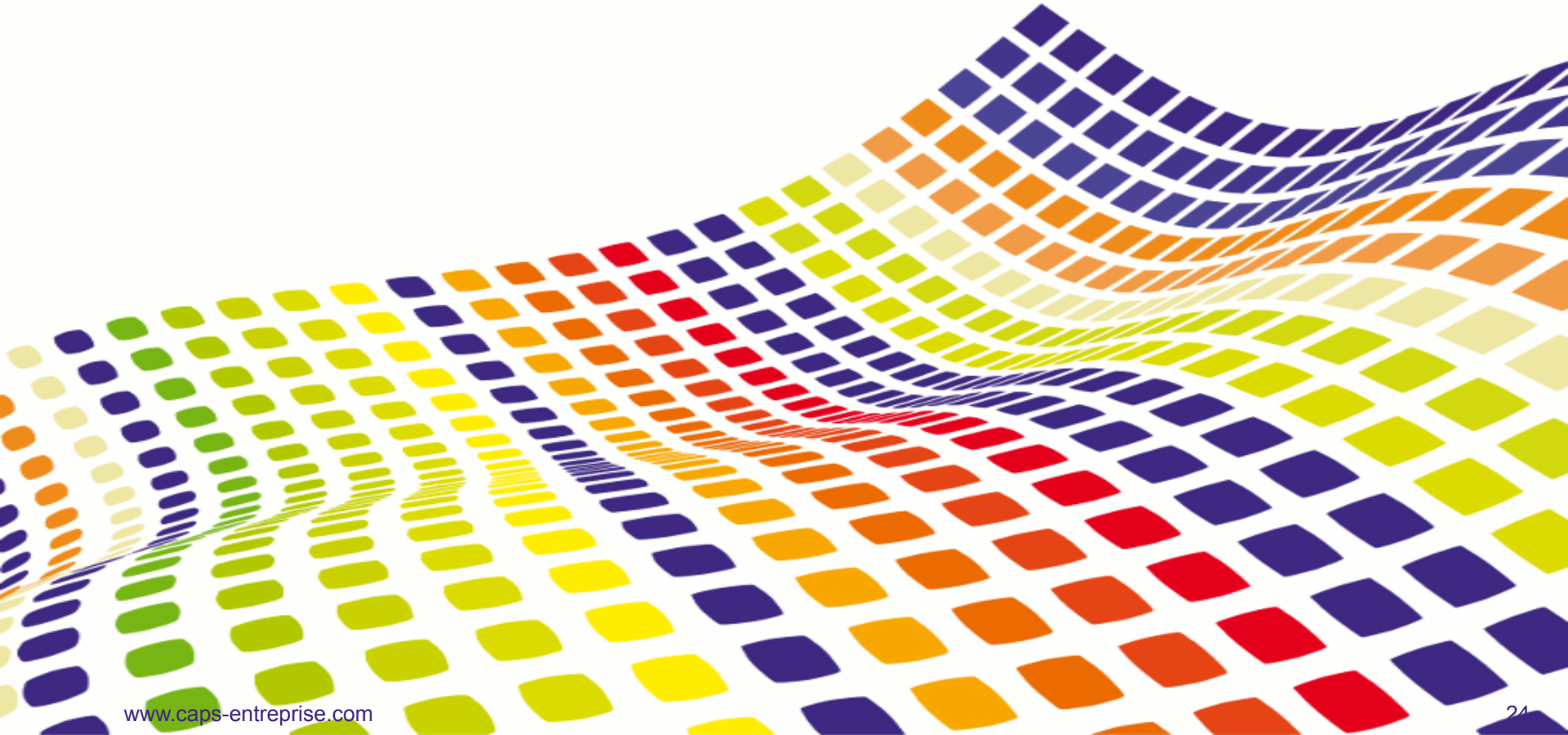
Ease application deployment

HMPP Runtime Library

- A lightweight software layer to adapt application to platform configuration and workload
  - A single binary running on various hardware platforms
  - Manage resource availability and codelet scheduling
- Ease application deployment in multi-GPUs systems
- Ensure application interoperability

Ease application deployment

Automatically Produce Hybrid Applications with an Open Tool

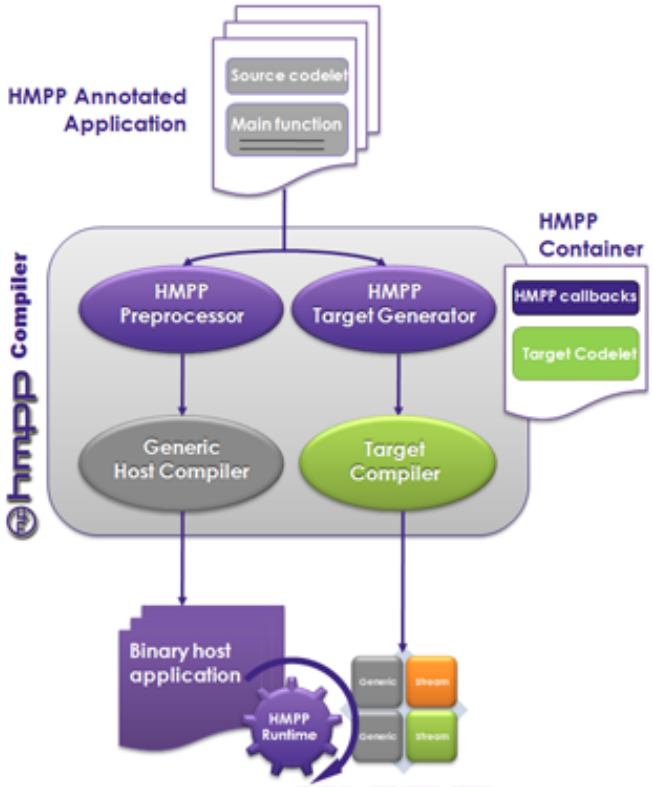


HMPP Compilation Passes

- HMPP drives all compilation passes

```
$ hmpp icc myProgram.c
```

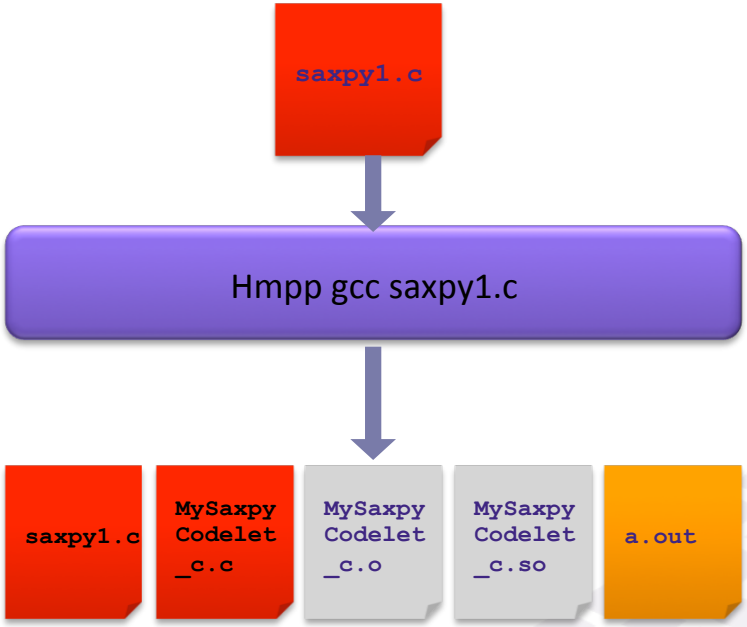
- Two main passes
  - Host application compilation
  - Codelet production



Open tool

HMPP Generated Files

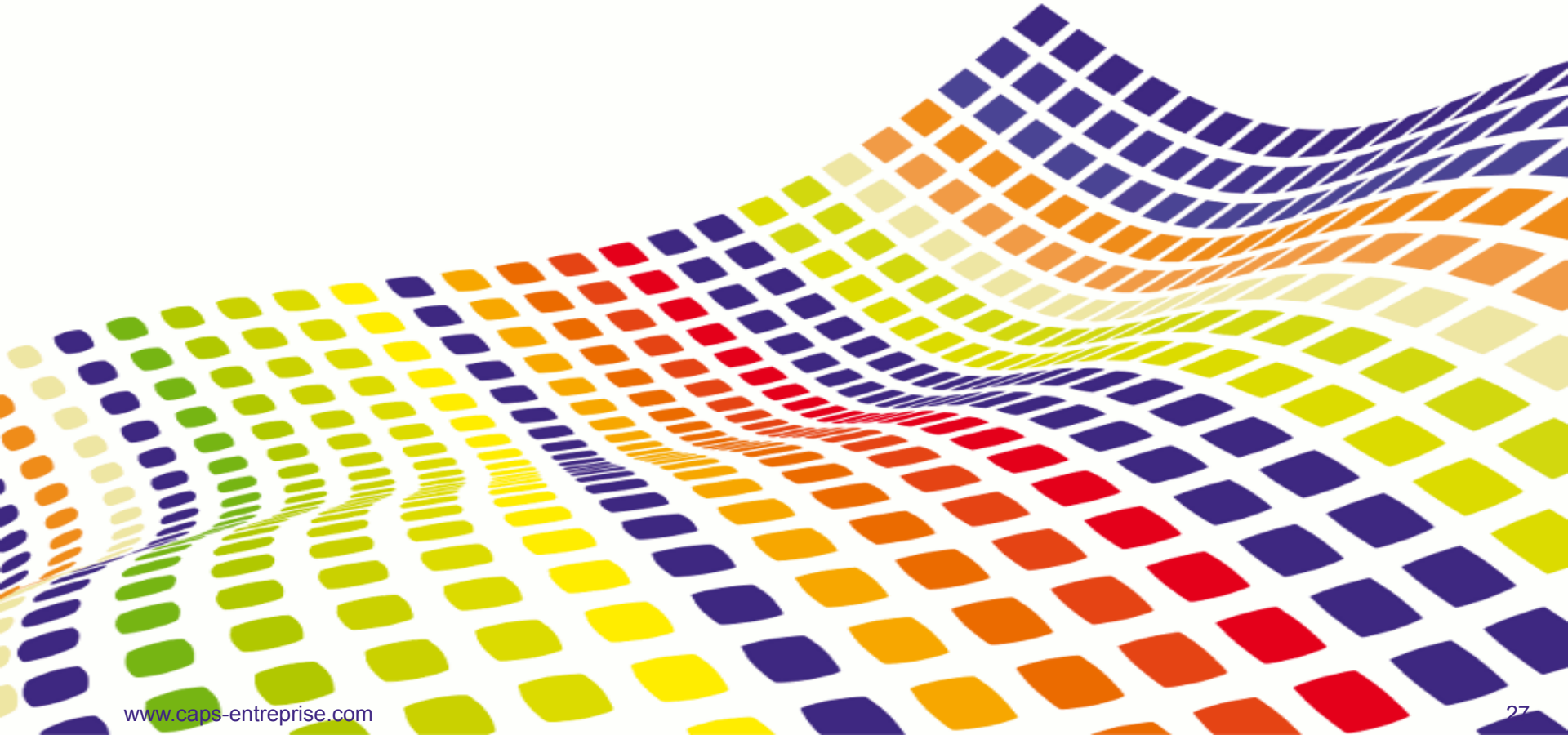
- Compiling generated codelet files
- All files are available for modification
  - Codelets and main source files




```
<codeletName>_<target>.<ext>
mySaxpy1_cuda.cu
mySgemm1_ocl.cc
...
```

Open tool

Reduce Development Time



HMPP Basic Programming

```
#pragma hmpp sgemm 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 sgemm callsite
    sgemm( size, size, size, alpha, vin1, vin2, beta, vout );
  }
 …
```

With only two directives!

Reduce development time

HMPP Abstracts the Hardware

CAPS

A set of advanced directives to reach performance

- **ADVANCEDLOAD** : Explicit data transfer CPU -> HWA
- **DELEGATEDSTORE** : Explicit data transfer HWA -> CPU
- **SYNCHRONIZE** : Wait for completion of the callsite
- **ALLOCATE** : Reserve HWA and allocate all data
- **RELEASE** : Release the allocated HWA
- **GROUP** : Define a group of codelets
- **MAP & MAPBYNAME** : Map arguments together
- **RESIDENT** : Declare a resident variable (global)

Reduce development time

HMPP Advanced Programming

```
int main(int argc, char **argv) {
…
#pragma hmpp sgemm allocate, args[vin1;vin2;vout].size={size,size}
#pragma hmpp sgemm advancedload, args[vin1;vin2;vout;m;n;k;alpha;beta]

  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
```

Allocate and initialize device outside loop

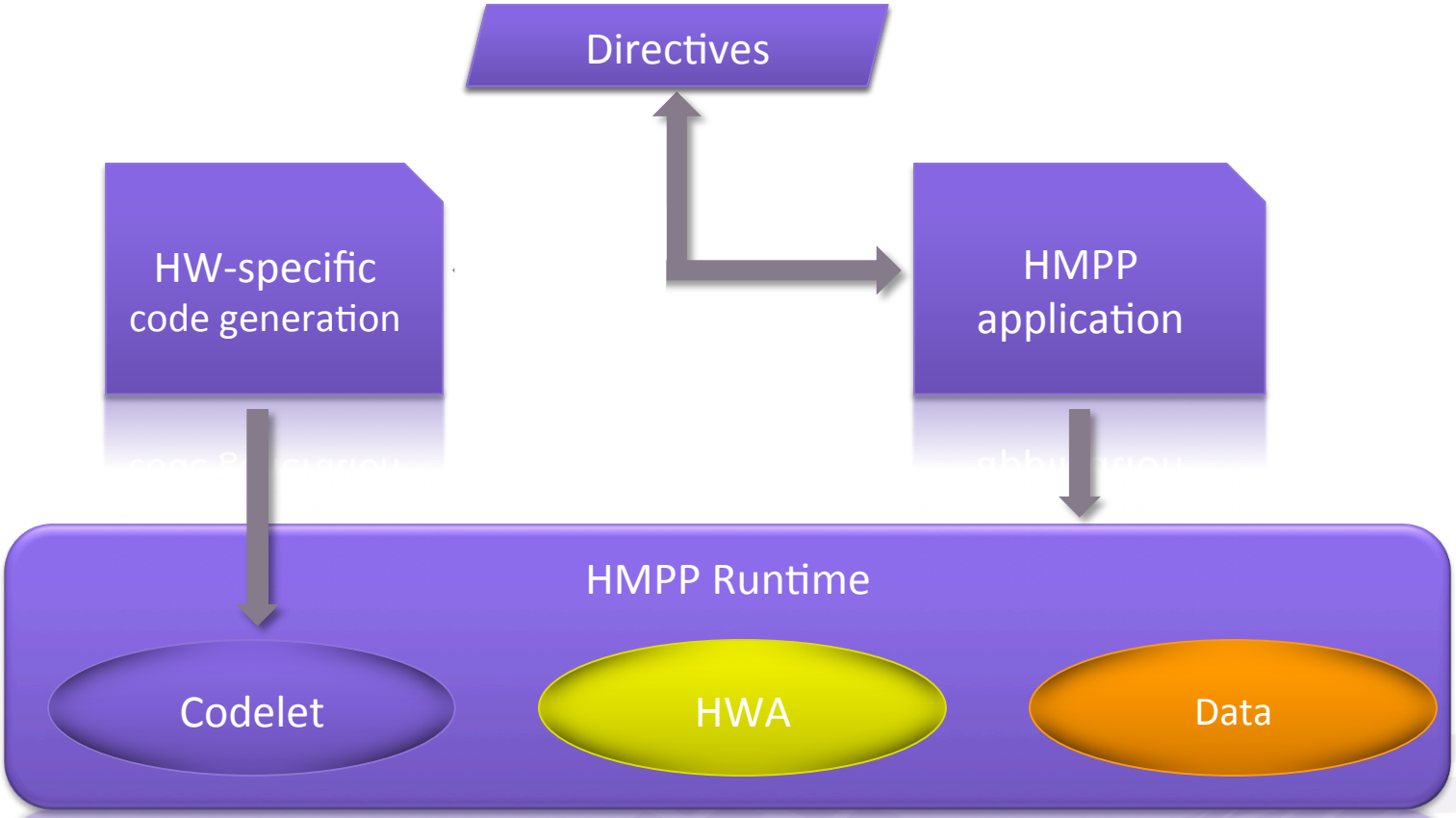
Preload data

Execute asynchronously

Download result when needed

Reduce development time

What about Directives for Code Generation?



Leverage Computing Power

Codelet Tuning Directives for High Level Optimization

CAPS

High level application tweaking

- By adding properties
  - (no)parallel, alias
- Using code transformation
  - Loop tiling, unroll, jam, permute, fuse, …
- Using target specific directives
  - Micro Architecture Management (warp size…)
  - Memory Management (CUDA shared memory, constant…)

Leverage Computing Power

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

Leverage Computing Power

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

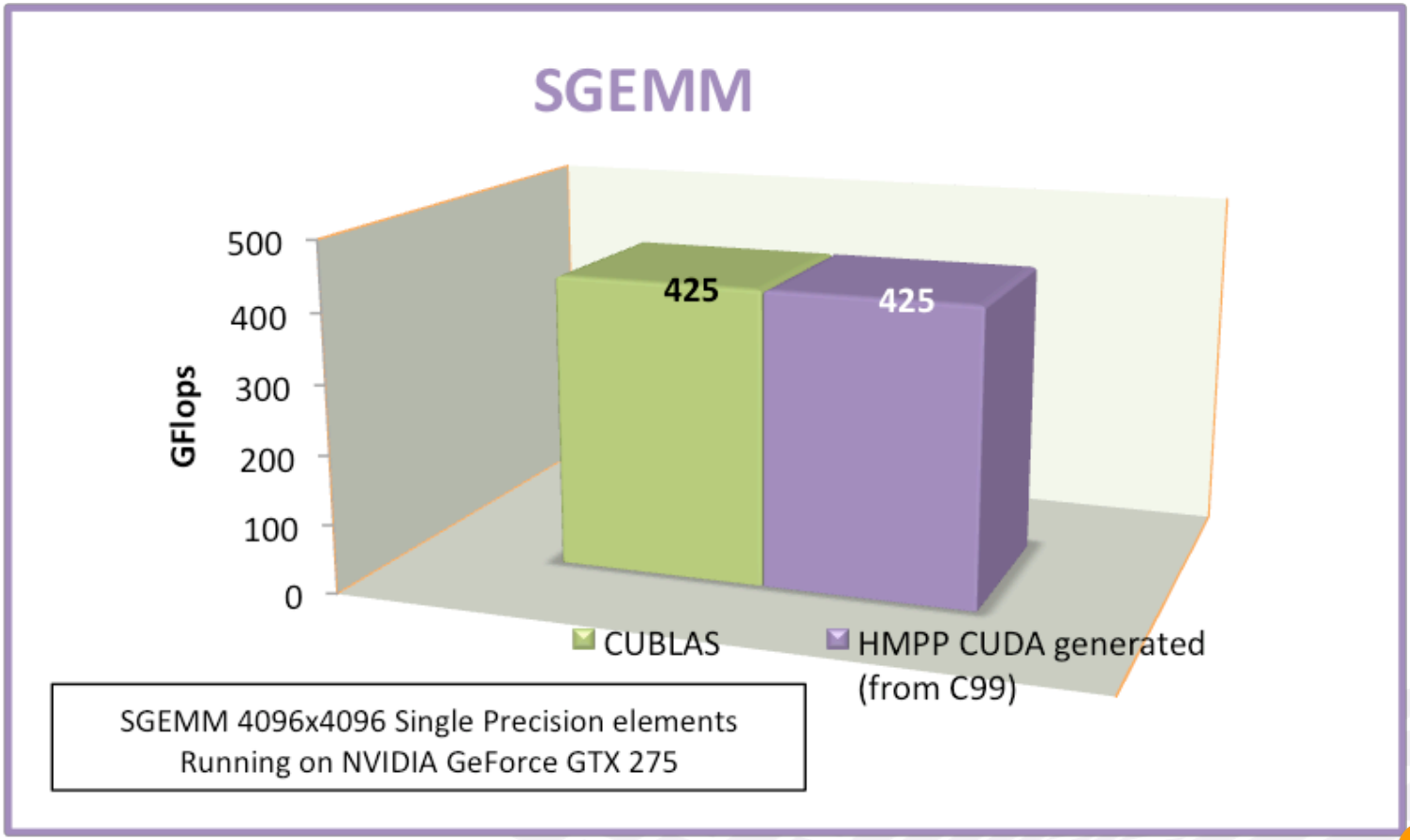
Detect grid support

Declare buf in shared memory

Parallel load in shared memory

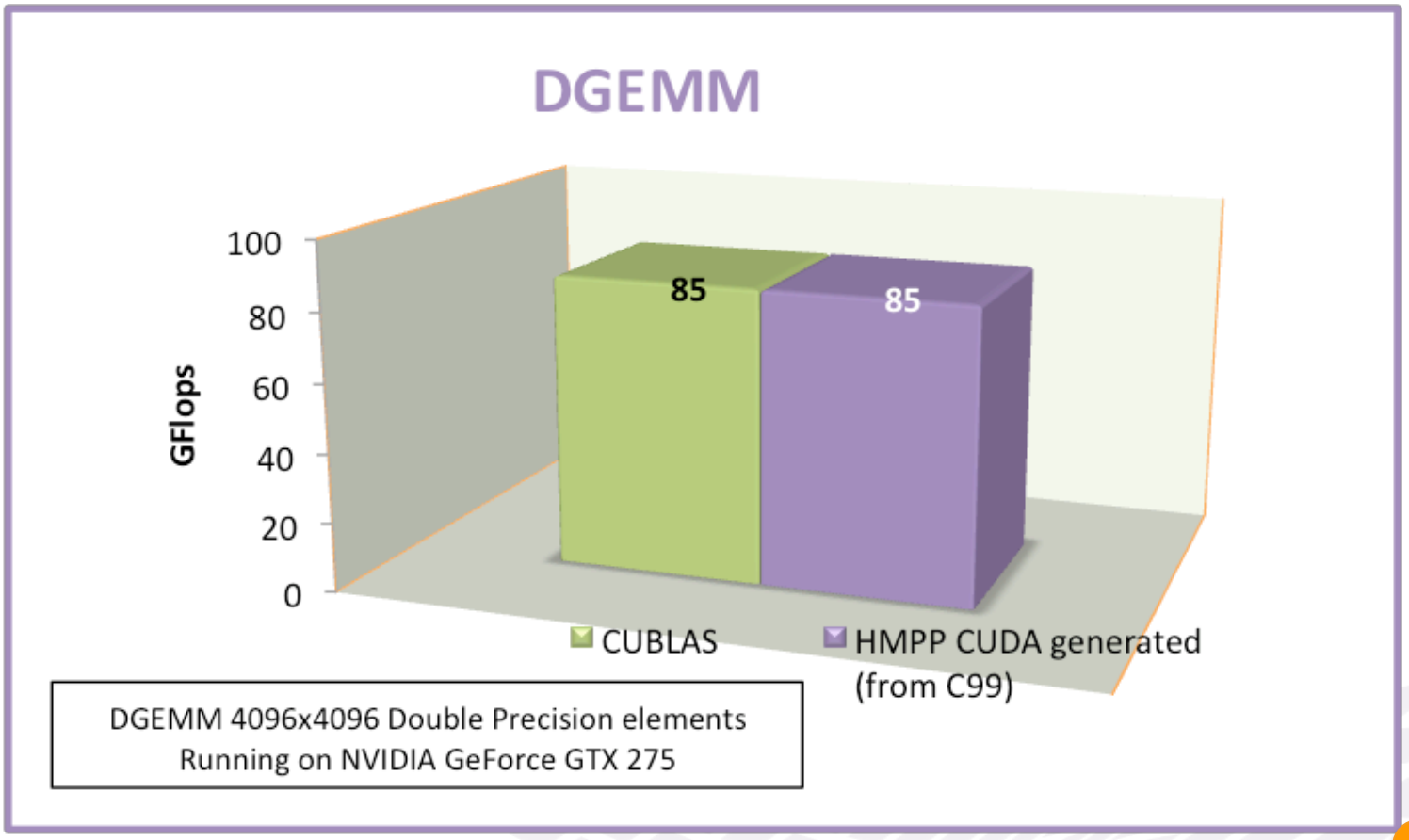
Wait for end loading before use

Generated Codelet Performance – advanced use



Leverage Computing Power

Generated Codelet Performance – advanced use

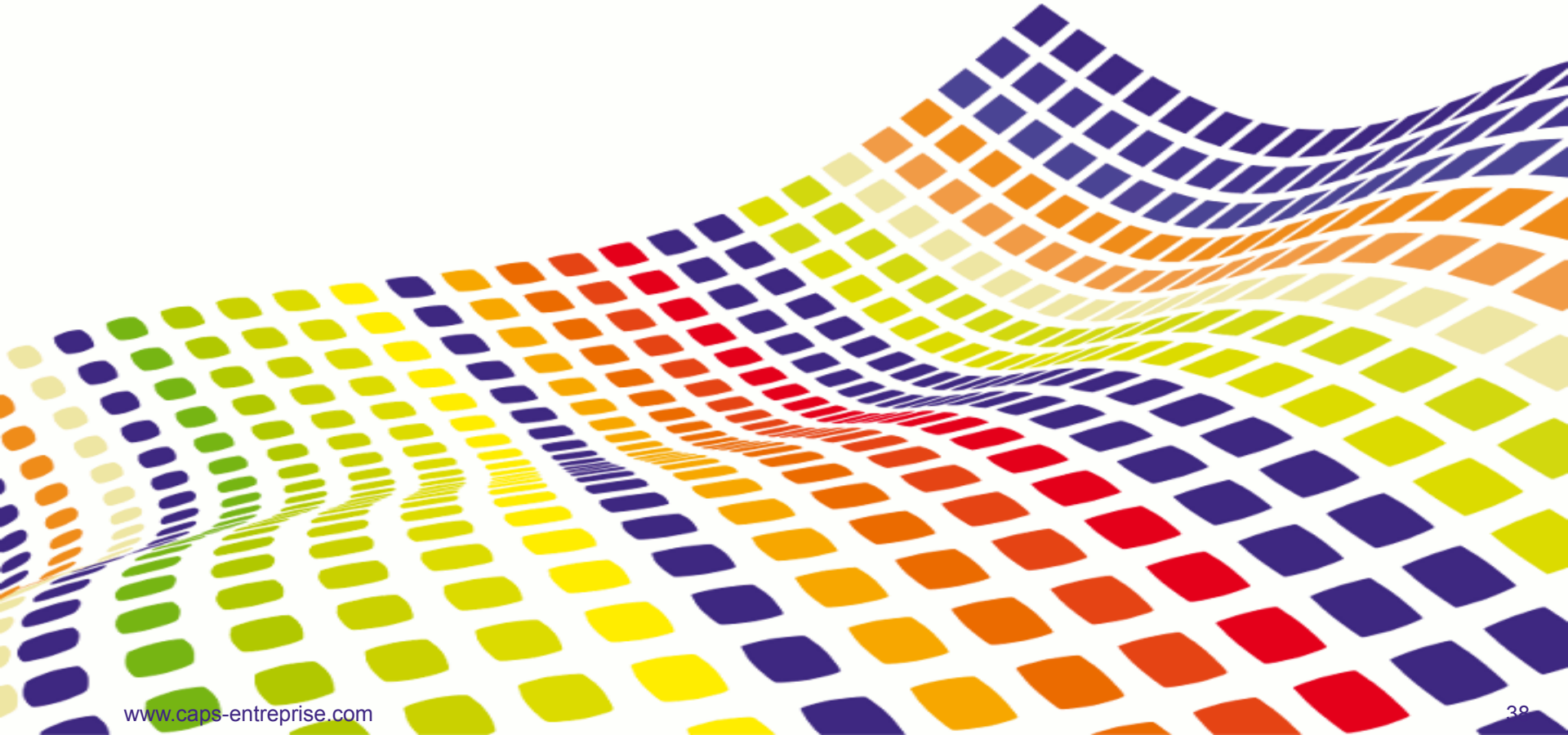


Leverage Computing Power

Conclusion

- Standardization effort is going on
  - OpenCL from Khronos
  - OpenMP accelerator subcommittee
  - HMPP as an Open Standard
- High level GPU code generation allows many GPU code optimization opportunities
  - Easier to tune applications at high level
- 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
  - Fine tuning is (unfortunately) specific to a GPU configuration

HMPP Features & Roadmap



HMPP features

- Sections (Partial transfers)
  - Split your transfers to fit with hardware
- Regions
  - Create a Codelet without any function!
- Complex numbers
- Reductions
- Hardware specific memories
  - Constant memory (CUDA)
  - Shared memory (CUDA)
- Native compilers
  - Icc, Ifort, Gcc, Gfortran…

What's next in HMPPCG?

- Native functions
  - Hand-written CUDA or OpenCL kernels provided by the user
- External functions
  - External C of Fortran functions called from the codelet
- Alternative implementations
  - Third-party optimized library functions instead of HMPP's generated code

Extern functions

- Support of function calls inside codelets
  - 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 of external definition

'sum' is an external function

Declare 'sum' as a function called in a codelet

Native functions

- Support of handwritten function calls inside codelets
  - Manual optimization (handwritten CUDA, OpenCL code)
  - For advanced users

native.c

```
float sum( float x, float y ) {
  return x+y;
}

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 native, sum

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

'sum' is an native function

native.xml

```
<hmppcg>
    <function name="sum">
        <signature language="c">
            float sum(float x, float y);
        </signature>

        <definition target="cuda">
        <![CDATA[
__device__ float sum(float x, float y) {
  return x+y;
}
         ]]>
         </definition>

        <definition target="opencl">
        <![CDATA[
float sum(float x, float y) {
  return x+y;
}
         ]]>
         </definition>
</function>
</hmppcg>
```

Native functions for different targets

Integration of external GPU libraries

```
!$hmppalt cula proxy

INTERFACE
!$hmppalt cula declare, name="sgeqrf" , extend(error,...), fallback = true
   SUBROUTINE hmpp_cula_sgeqrf(err, m, n, a, lda, tau, work, lwork, info)
     INTEGER, INTENT(INOUT) :: err
     INTEGER, INTENT(IN)    :: m, n, lda,lwork
     INTEGER, INTENT(OUT)   :: info
     REAL(4), INTENT(IN)    :: a(lda, *), tau(*)
     REAL(4), INTENT(INOUT) :: work(*)
   END SUBROUTINE hmpp_cula_sgeqrf
…

END INTERFACE
```

Declaration of the proxy

Declaration of the proxy function

```
…
  ! Regular call of the CPU version of sgeqrf
  CALL sgeqrf(M, N, A, LDA, TAU, WORK, LWORK, INFO);

  ! Implement sgeqrf using CULA for large matrices only

  !$HMPPALT CULA call, name="sgeqrf", cond="M*N>1000000"
  CALL sgeqrf (M, N, A2, LDA, TAU2, WORK, LWORK, INFO)
…
```

HMPP proxy call

What's next in HMPP?

- Input code
  - C
  - Fortran 90
  - C++ (coming soon)
- Targeted accelerators
  - CUDA (Nvidia)
  - OpenCL (AMD/ATI, Nvidia)
- Targeted Operating Systems
  - Linux
  - Windows

CAPS

Innovative Software for Manycore Paradigms