Programming ManyCore Systems

CAPS OpenACC Compilers

Stéphane BIHAN
CERN – openlab
2013, 27th of Feb.

About CAPS

CAPS Mission and Offer

Mission
Help you break the parallel wall and
harness the power of manycore computing

- Software programming tools
  - CAPS Manycore Compilers
  - CAPS Workbench: set of development tools
- CAPS engineering services
  - Port applications to manycore systems
  - Fine tune parallel applications
  - Recommend machine configurations
  - Diagnose application parallelism
- Training sessions
  - OpenACC, CUDA, OpenCL,
  - OpenMP/pthreads

Based in Rennes, France
Created in 2002
More than 10 years of expertise
About 30 employees

CAPS Worldwide

Headquarters - FRANCE
Immeuble CAP Nord
4A Allée Marie Berhaut
35000 Rennes
France
Tel.: +33 (0)2 22 51 16 00
info@caps-entreprise.com

CAPS - USA
26 O'Farell St; Suite 500
San Francisco
CA 94108
usa@caps-entreprise.com

CAPS - CHINA
Suite E2, 30/F
JuneYao International Plaza
789, Zhaojiabang Road,
Shanghai 200032
Tel.: +86 21 3363 0057
apac@caps-entreprise.com

CAPS Ecosystem

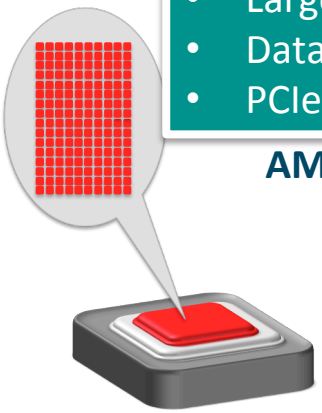
They trust us

Resellers and Partners

- ✓Main chip makers
- ✓Hardware constructors
- ✓Major software leaders

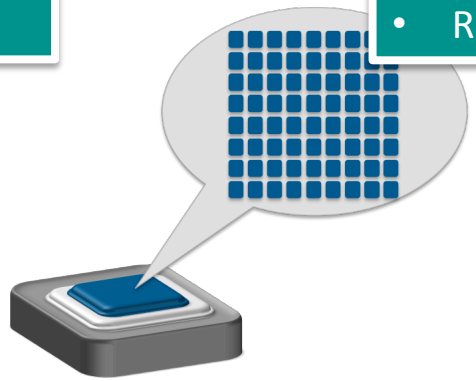
Many-Core Technology Insights

Various Many-Core Paths



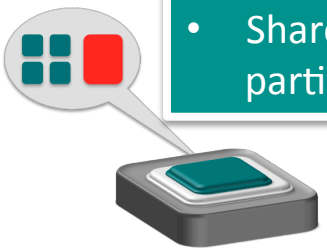
- Large number of small cores
- Data parallelism is key
- PCIe to CPU connection

AMD Discrete GPU



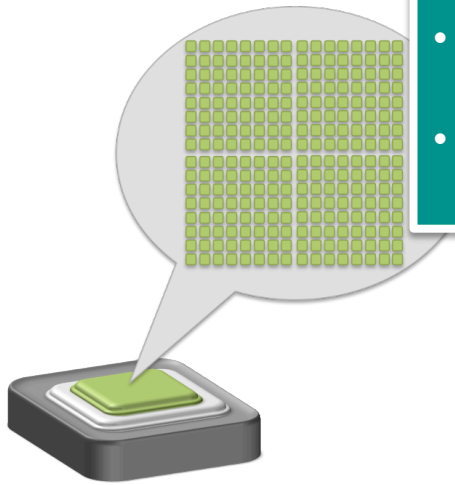
- 50+ number of x86 cores
- Support conventional programming
- Vectorization is key
- Run as an accelerator or standalone

INTEL Many Integrated Cores



- Integrated CPU+GPU cores
- Target power efficient devices at this stage
- Shared memory system with partitions

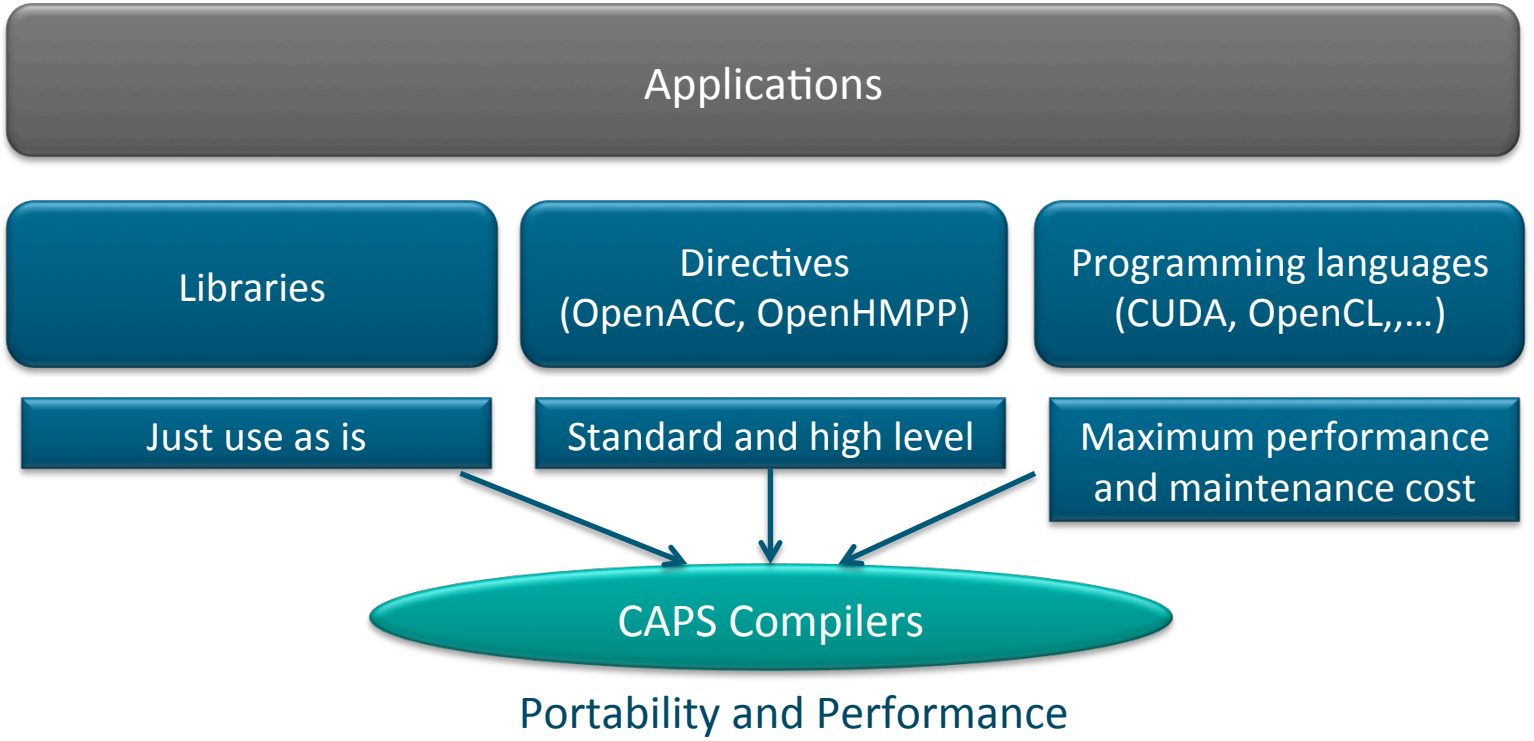
AMD APU

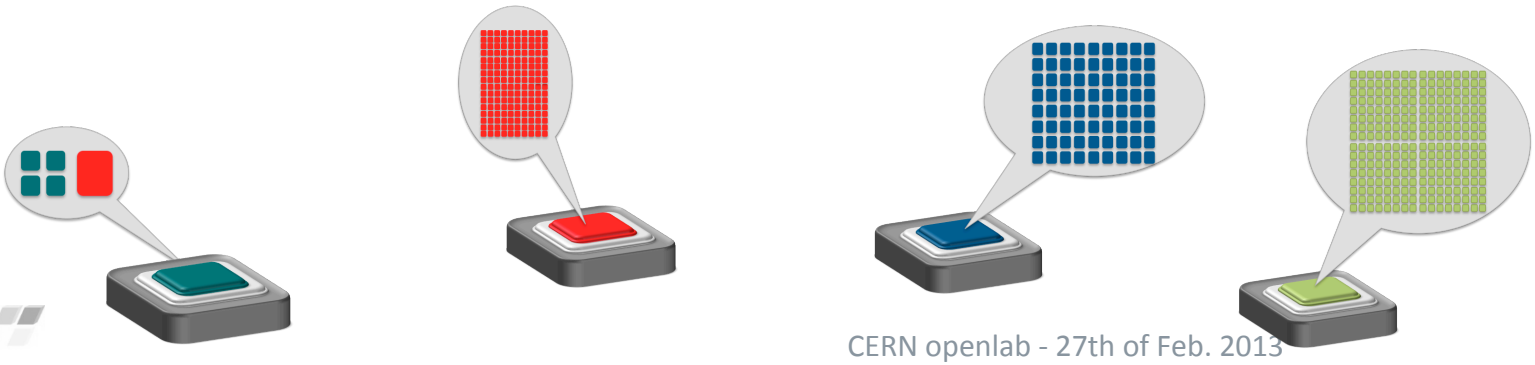


- Large number of small cores
- Data parallelism is key
- Support nested and dynamic parallelism
- PCIe to host CPU or low power ARM CPU (CARMA)

NVIDIA GPU

Many-core Programming Models





Directive-based Programming

- Consortium created in 2011 by CAPS, CRAY, NVIDIA and PGI
  - New members in 2012: Allinea, Georgia Tech, ORNL, TU-Dresden, University of Houston, Rogue Wave
  - Momentum with broad adoption
  - 2nd specification to be announced

OpenACC
DIRECTIVES FOR ACCELERATORS

- Created by CAPS in 2007 and adopted by PathScale
  - Advanced features
    - Multi devices
    - Native and accelerated libraries with directives and proxy
    - Tuning directives for loop optimizations, use of device features, …
    - Ground for innovation in CAPS
  - Can interoperate with OpenACC

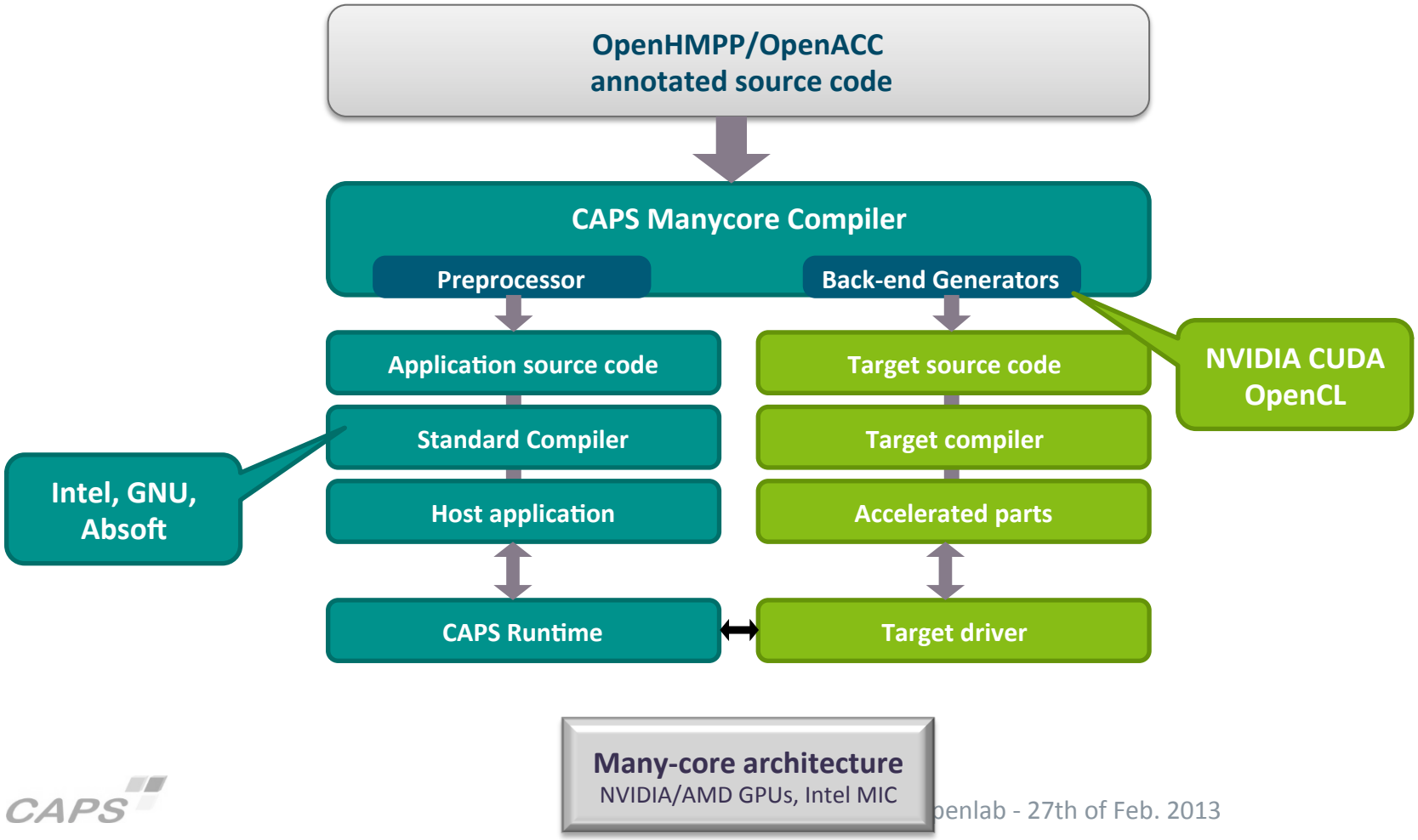
open hmpp

- First technical report proposing directives for accelerators
  - Not a specification yet
  - Aim is to get early feedback

OpenMP

CAPS Compilers

- Source to source compilers
  - Use your preferred native x86 and hardware vendor compilers

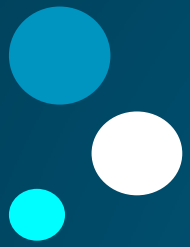


How to Use CAPS Compiler

- Here is an example with an OpenACC compiler from CAPS:

```
$ capsmc gcc myopenaccapp.c -o myopenaccapp.exe
```

- Host compatible compilers :
  - GNU
  - Open64
  - Intel
  - Absoft
  - …



OpenACC and OpenHMPP Directive-based Programming

Simply Accelerate with OpenACC

```
#pragma acc kernels
{
#pragma acc loop independent
  for (int i = 0; i < n; ++i){
    for (int j = 0; j < n; ++j){
      for (int k = 0; k < n; ++k){
        B[i][j*k%n] = A[i][j*k%n];
      }
    }
  }
```

All loop nests in a *kernels* construct will be a differentkernel

Each loop nest is launched in order in the device

Add independent clause to disambiguate pointers

The entire *parallel* construct becomes a single kernel

```
#pragma acc parallel {
  #pragma acc loop
  for (int i = 0; i < n; ++i){
    #pragma acc loop
    for (int j = 0; j < n; ++j){
      B[i][j] = A[i][j];
    }
  }
  for(int k=0; k < n; k++){
    #pragma acc loop
    for (int i = 0; i < n; ++i){
      #pragma acc loop
      for (int j = 0; j < n; ++j){
        C[k][i][j] = B[k-1][i+1][j] + …;
  } } }
}
```

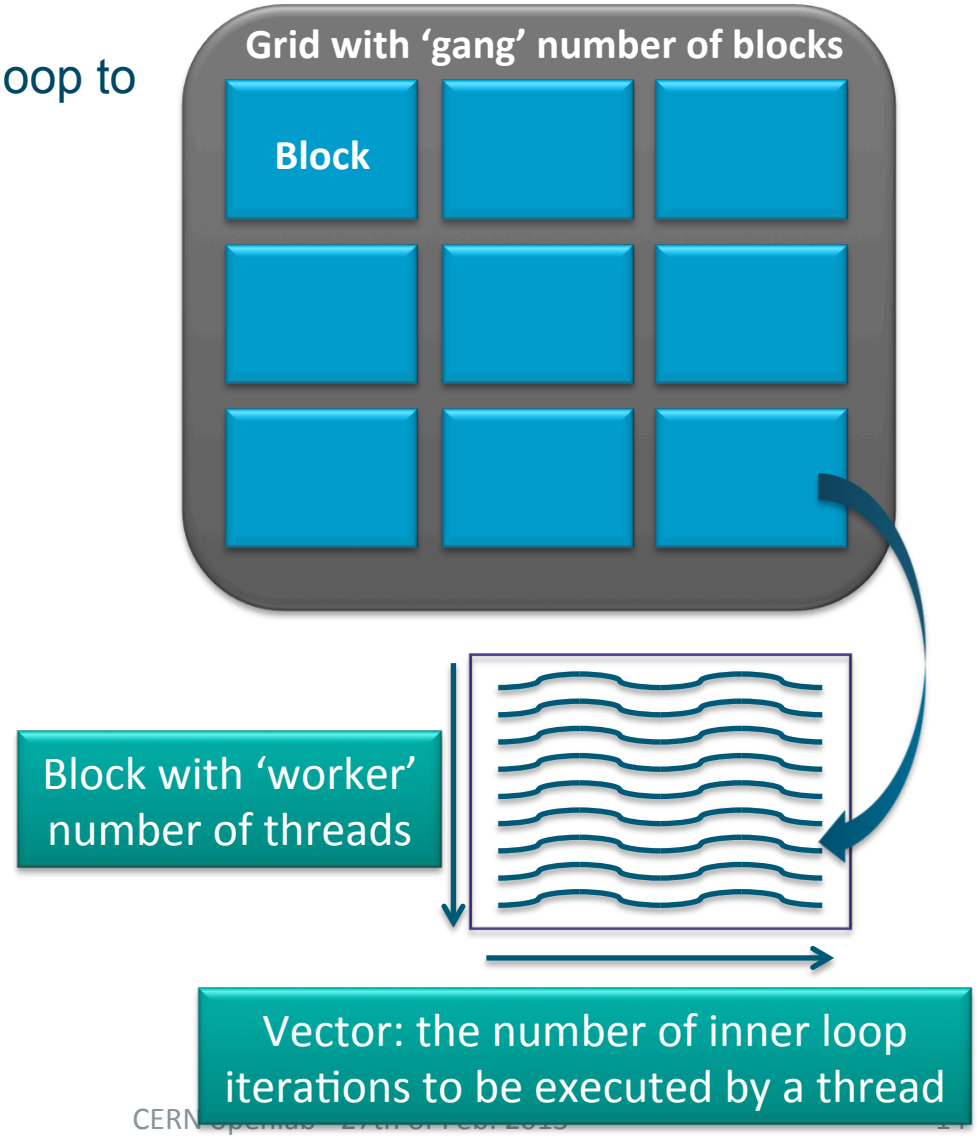
Add the loop directives to distribute the work amongst the threads (redundant otherwise)

CAPS

Gang, Workers and Vector in a Grid

Either let the compiler map the parallel loop to the accelerator or configure manually

```
#pragma acc kernels
{
#pragma acc loop independent
  for (int i = 0; i < n; ++i){
    for (int j = 0; j < n; ++j){
      for (int k = 0; k < n; ++k){
        B[i][j*k%n] = A[i][j*k%n];
      }
    }
  }
#pragma acc loop gang(NB) worker(NT)
  for (int i = 0; i < n; ++i){
    #pragma acc loop vector(NI)
    for (int j = 0; j < m; ++j){
      B[i][j] = i * j * A[i][j];
    }
  }
}
```



OpenHMPP Codelet/Callsite Directives

- Accelerate function calls only

```
#pragma hmpp sgemm codelet, target=CUDA:OPENCL, args[vout].io=inout, &
#pragma hmpp & args[*].transfer=auto
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 );
  }
    /* . . . */
 }
```

Declare CUDA and OPENCL codelets

Synchronous codelet call

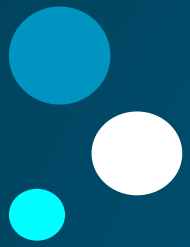
Managing Data

Data construct region with In/out

Explicit copy from device to host

Use *create* clause in data construct for accelerator-local arrays

```
#pragma acc data copyin(A[1:N-2]),
  copyout(B[N])
{
  #pragma acc kernels
  {
    #pragma acc loop independent
    for (int i = 0; i < N; ++i){
      A[i][0] = ...;
      A[i][M - 1] = 0.0f;
    }
    ...
  }
  #pragma acc update host(A)
    ...
  #pragma acc kernels
  for (int i = 0; i < n; ++i){
      B[i] = ...;
  }
}
```



OpenHMPP Advanced Programming

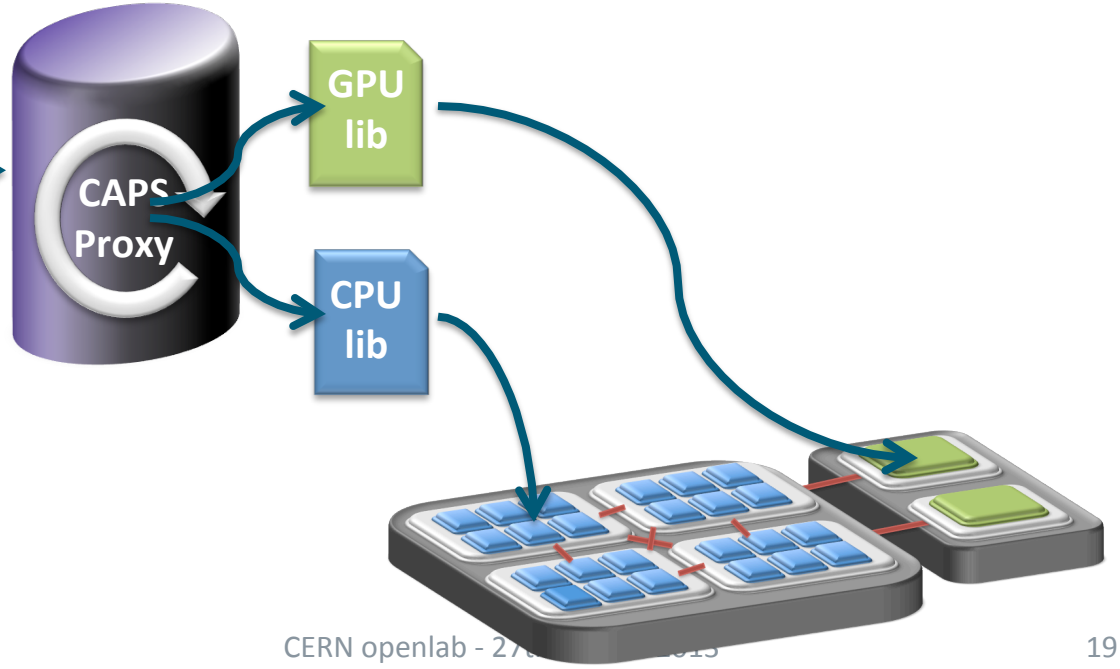
What's in OpenHMPP not in OpenACC?

- Use specialized libraries with directives
- Support for multiple devices
- Zero-copy transfers on APUs
- Tuning directives
- Use native and external kernels

Using Accelerated Libraries with OpenHMPP

- Single OpenHMPP directive before call to library routine
  - Preserve original source code
  - Make CPU and specialized libraries co-exist through a proxy
  - Use most efficient library depending on data size for instance

```
...
...
...
...
#pragma hmppalt
call CPU_blas(…)
...
...
call CPU_fft(…)...
...
```



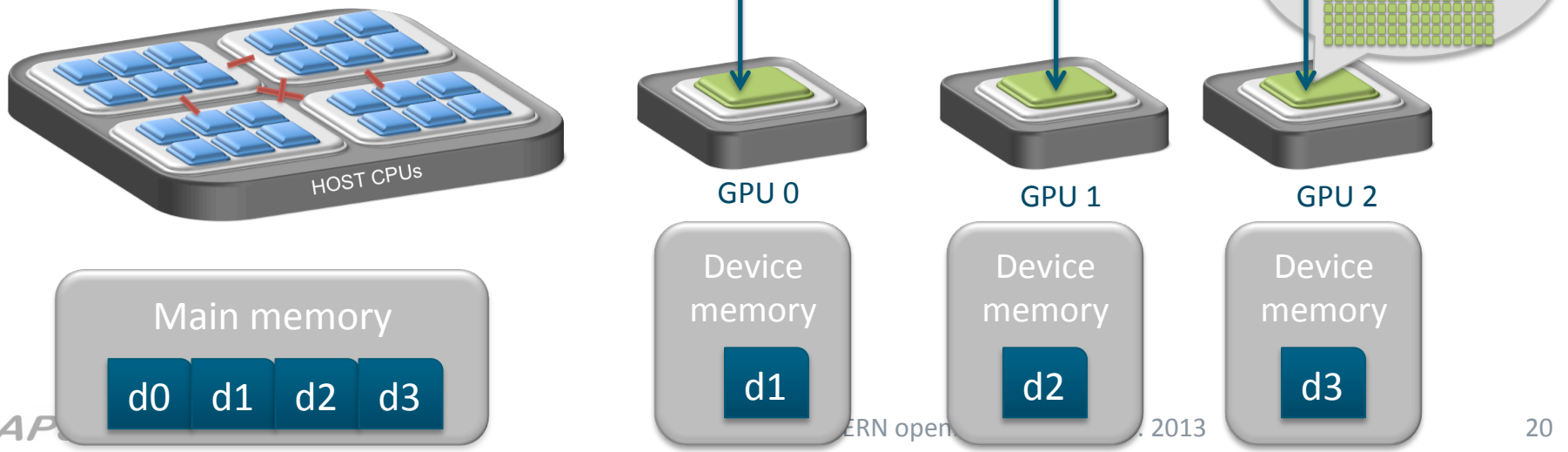
Multi-GPU and Data Collection

```
#pragma hmpp parallel, device="i%2"
  for( i=0;i<n;i++){
    //Allocate the mirrors for d
    #pragma hmpp <MyGroup> allocate, data[”d[i+1]”
...
  }
```

```
#pragma hmpp <MyGroup> parallel
for(k=0;k<n;k++) {
     #pragma hmpp <MyGroup> f1 callsite
         myparallelfunc(d[k],n);
   }
```

Distribute the data over the accelerators

Execute the tasks in parallel according to the owner compute rule



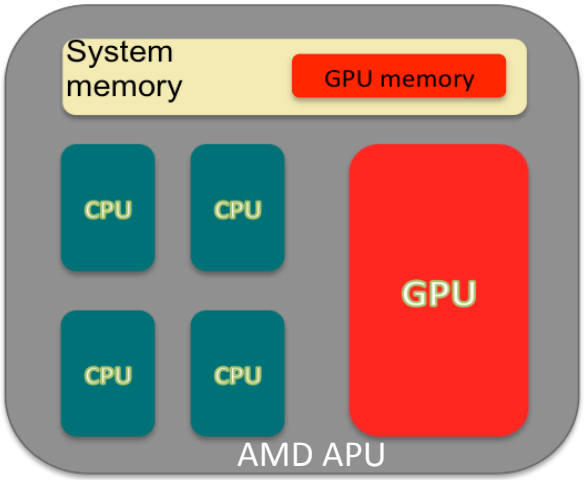
Zero-copy Transfers on AMD APU

- AMD APUs have a single system memory with a GPU partition
  - Avoid duplication of memory objects between CPU and GPU
  - Avoid copies required for memory coherency between CPU and GPU
  - Minimization of data movement not required anymore to get performance

```
#pragma hmpp replacealloc="cl_mem_alloc_host_ptr"
  float * A = malloc(N * sizeof(float));
  float * B = malloc(N * sizeof(float));
#pragma hmpp replacealloc="host”
```

```
void foo(int n, float* A, float* B){
  int i;

#pragma acc kernels, pcopy(A[:n],B[:n])
#pragma acc loop independent
  for (i = 0; i < n; ++i)
  {
#pragma hmppcg memspace openclalloc (A,B)
    A[i] = -i + A[i];
    B[i] = 2*i + A[i];
  }
}
```



OpenHMPP Tuning Directives

- Apply loop transformations
  - Loop unrolling, blocking, tiling, permute, …
- Map computations on accelerators
  - Control gridification

```
#pragma hmppcg(CUDA) gridify(j,i), blocksize "64x1"
#pragma hmppcg(CUDA) unroll(8), jam, split, noremainder
 for( i = 0 ; i < n; i++ ) {
   int j;
#pragma hmppcg(CUDA) unroll(4), jam(i), noremainder
   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

OpenHMPP Tuning Directives (2)

- Use hardware specific features
  - constant and shared memory, NVIDIA UVA, ...
  - GPU synchronization barriers

```
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 gridify(i), blocksize 256x1, shared(buf)
for (i=DIST; i<N-DIST ; i++){
   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

External Functions Directives

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;
}
```

Declare ‘sum’ as a function to be called in a codelet

Extern.c

Import external definition

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

‘sum’ is an external function

```
$ hmpp --function=hmppcg_functions.dpil gcc source_files
```

Native CUDA/OpenCL Functions Support

- Integration of handwritten CUDA and OpenCL codes with directives

kernel.c

```
float f1 ( float x, float y,
           float alpha, float pow)
{
  float res;
  res = alpha * cosf(x);
  res += powf(y, pow);
  return sqrtf(res);
}

#pragma hmpp mycdlt codelet, …
void myfunc( int size, float alpha,
             float pow, float *v1,
             float *v2, float *res)
{
  int i;
  float t[size],temp;
#pragma hmppcg include (<native.cu>)
#pragma hmppcg native (f1)
  for(i = 0 ; i < size ; i++ )
    res[i] = f1( v1[i],v2[i],
                 alpha, pow);
}
…
```

native.cu

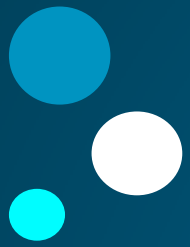
```
__device__ float f1 ( float x, float y,
                      float alpha, float pow)
{
  float res;
  res = alpha * __cosf(x);
  res += __powf(y, pow);
  return __fsqrt_rn(res);
}
```

Native function in CUDA

Include file containing native function definition

Use a GPU sqrt

'f1' is a native function

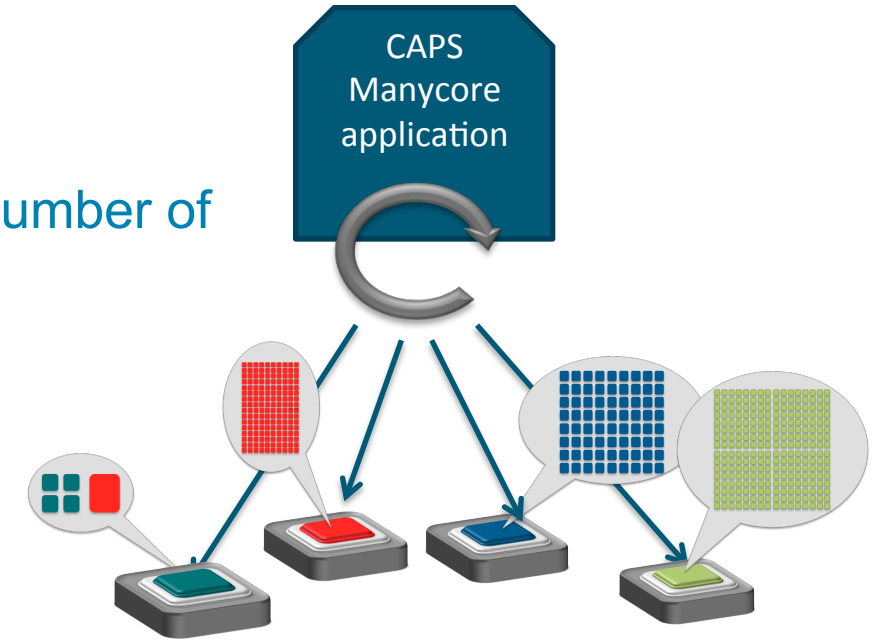


Auto Tuning with CAPS Compilers

Auto Tuning Principles

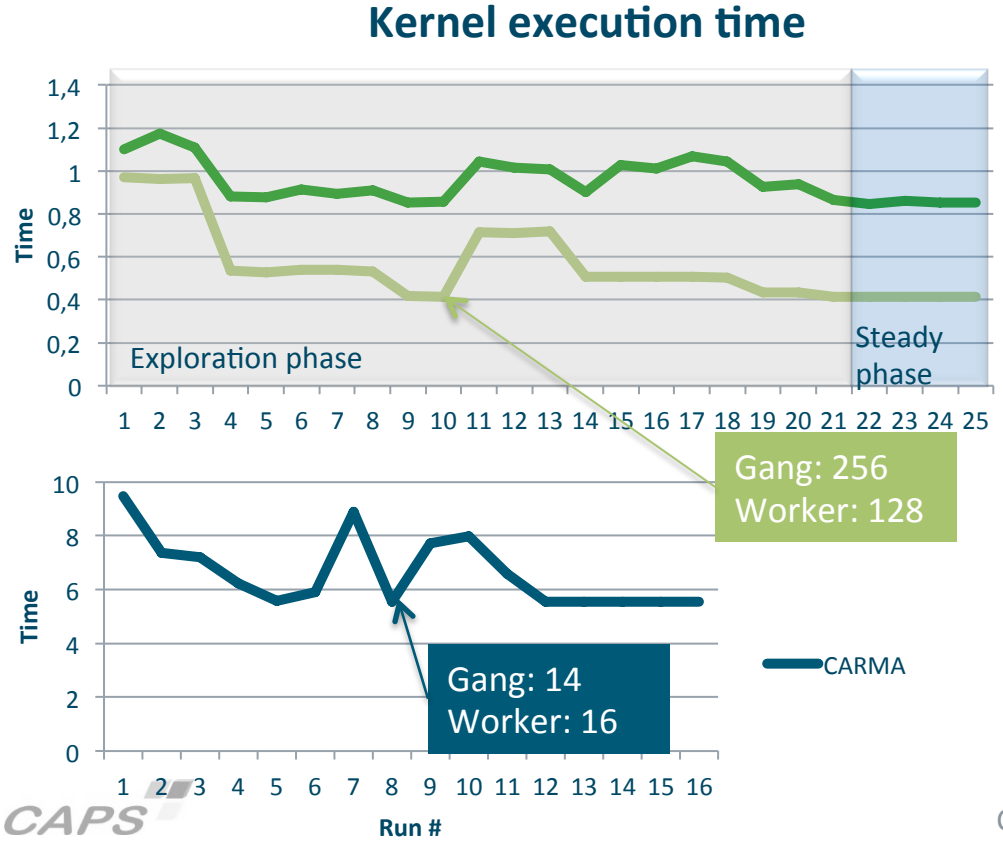
- Make applications dynamically adapt to various accelerator architectures
  - Find efficient mapping of kernels on computational grid
  - Different types of accelerators (NVIDIA/AMD GPUs, Intel MIC, …)
  - Different versions of architectures (Fermi, Kepler)

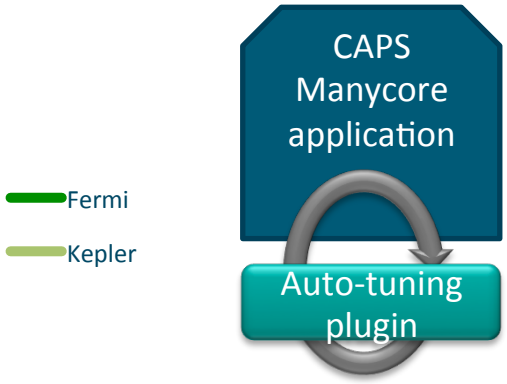
- Explore optimization space
  - Determine architecture-efficient number of gangs, workers and vector size
  - Select optimum kernel variant

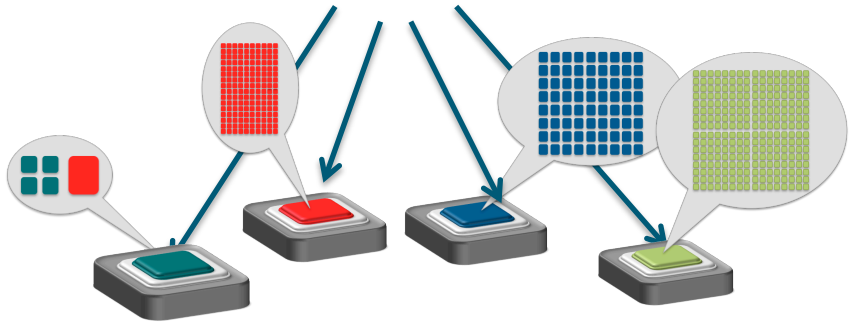


CAPS Auto-tuning Mechanism

- Auto-tuning plugin
  - Select different number of gangs/workers or kernel variants at runtime
  - Execute and record kernel performance within the application
  - Once exploration has complete, uses best found values for other executions







Dynamic Mapping of Kernels on a Grid

Gang and worker values to explore

```
size_t gangsVariant[] =   {  14, 14, 14, 14, 14 , 14  ,14 , 14,  28, 28, 28, 28,
28, 28 , 28,  28};
  size_t workersVariant[] = {  2,  4,  6,   8, 10,  12,  14,  16,   2,  4,  6,  8,
10, 12 , 14,  16};

  int filterVariantSelector = variantSelectorState(__FILE__ ":32", 15);

  int gangs = gangsVariant[filterVariantSelector];
  int workers = workersVariant[filterVariantSelector];

#pragma acc parallel                                                    \
  copy(h_vvs[0:spp * spp]),                                             \
  copyin( h_sitevalues[0:spp * endsite * 4],  weightrat[0:endsite])     \
  num_gangs(gangs), num_workers(workers), vector_length(32)
  {
    int i,j,x,y;

    TYPE sitevalue10;
    TYPE sitevalue11;
    TYPE sitevalue12;
```

Retrieve variants' value

Set variant value

Kernel Variant Selection

Define variants of kernels

```
void filterStencil5x5_V2(const uint32 p_heigh[1],
     const uint32 p_width[1], const RasterType filter[5][5],
     co
. . .
#pragma
#pragma
  for (
    ...
```

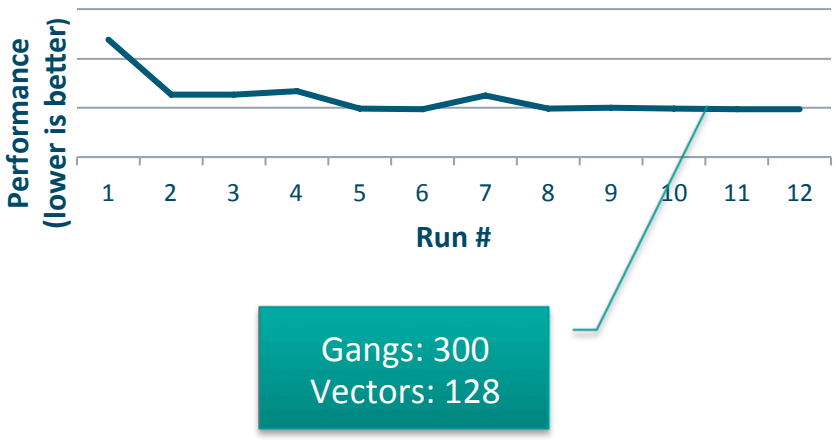
```
void filterStencil5x5_V1(const uint32 p_heigh[1],
     const uint32 p_width[1], const RasterType filter[5][5],
     const RasterType *p_inRaster, RasterType *p_outRaster){
. . .
#pragma hmppcg grid blocksize "32x4"
#pragma hmppcg unroll 6, jam
  for (i = stencil; i < heigh - stencil; i++) {
    ...
```

Call them through CAPS auto-tuning plugin

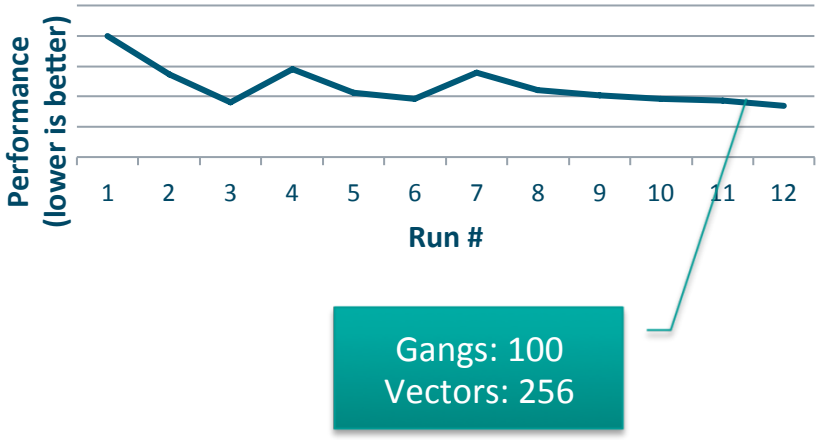
```
#pragma hmpp <convolution> filter5x5 callsite variants( &
#pragma hmpp & filterStencil5x5@<convolution>[C], &
#pragma hmpp & filterStencil5x5_V1@<convolution>[CUDA], &
#pragma hmpp & filterStencil5x5_V2@<convolution>[CUDA]) &
#pragma hmpp & selector(filterVariantSelector)
    filterStencil5x5(&fullHeigh, &width, stencil1, raster1, raster2);
```

Gang and Vector Auto-tuning Results

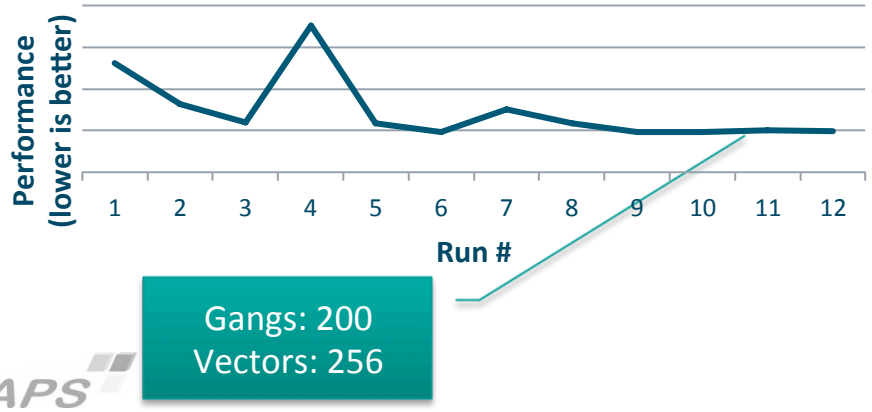
NVIDIA Kepler



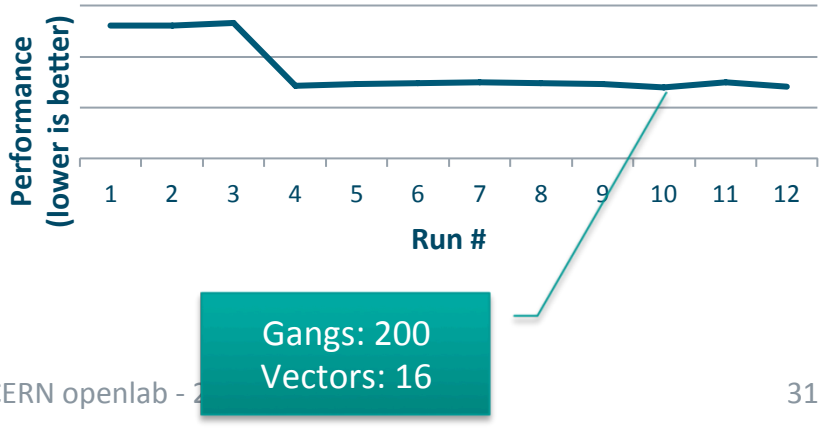
AMD Trinity APU

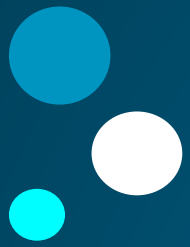


AMD Radeon



Intel MIC





One OpenCL to Rule Them All?

Objectives and Experimentation

- Objective: to study the performance portability of a unique code base
  - Or how much performance do we loose across different accelerators?
  - Tradeoff between performance and portability
- Experimental conditions
  - HydroC application
    - Mini application built from RAMSES code
    - 22 OpenACC kernels generated with CAPS Compiler OpenCL
      - Vary the thread grid configuration
      - Performance relative to the best performance
    - Communications are an insignificant part of total execution time
  - Multiple targets
    - NVIDIA K20C: 1170 GFlops DP
    - Intel Xeon Phi: 1070 GFlops DP
    - AMD 7970 GPU: 947 GFlops DP

Efficiency Loss Results (Lower is better)

WG	AMD 7970	K20C	SE10P	Avg.	Max.
16x4	0,67%	12,08%	23,56%	14,12%	23,56%
16x8	5,00%	5,75%	34,71%	15,16%	34,71%
16x16	7,54%	2,34%	38,44%	16,11%	38,44%
32x2	4,57%	22,20%	8,14%	11,64%	22,20%
32x4	2,95%	12,44%	10,96%	8,78%	12,44%
32x8	0,00%	5,89%	18,07%	7,99%	18,07%
64x1	9,64%	33,73%	3,54%	15,64%	33,73%
64x2	9,27%	20,38%	4,52%	11,39%	20,38%
64x4	4,64%	13,24%	5,78%	7,89%	13,24%
128x1	13,22%	27,31%	0,72%	13,75%	27,31%
128x2	7,53%	20,51%	2,15%	10,07%	20,51%
256x1	9,51%	28,56%	0,00%	12,69%	28,56%
16x32	N/A	0,00%	43,47%	21,74%	43,47%

Conclusion

Conclusion

- Many-Core becomes ubiquitous
  - Various accelerator architectures
  - Still as co-processors but toward on-die integrated
- Programming models converge
  - Momentum with OpenACC directive-based standard
  - Key point is portability
- New auto-tuning mechanisms
  - A way to achieve performance with portability (not portable performance!)
  - Portable performance is a trade-off
    - How much you are willing to loose in term of performance
    - Versus the effort to fine tune

CAPS Compilers

CAPS Compilers

	Workstation License			Server License
	Starter Edition	**Medium Edition**	**Full Edition**	**Full Edition**
Front-ends (C/Fortran/C++)	C or Fortran	C, Fortran or C++	All	All
Programming Directives (OpenHMPP/OpenACC)	OpenACC	All	All	All
Targets (CUDA/OpenCL)	CUDA or OpenCL	CUDA or OpenCL	All	All
License Type	User-based Node-locked	User-based Node-locked	User-based Node-locked	Floating
Support	Bronze	Silver	Gold	Gold
Price	199,00 €	399,00 €	999,00 €	2990,00 € (per token) 1290,00 € EDU

Available on store.caps-entreprise.com
stephane.bihan@caps-entreprise.com

Visit CAPS knowledge base http://kb.caps-entreprise.com/

Accelerator Programming model

Parallelization

CAPS Workbench

Directive-based programming

HPC

Portability

OpenCL

Parallel Computing

CAPS Compilers

Many-Core programming

NVIDIA CUDA

Code speedup

OpenHMPP

GPGPU

OpenACC

High Performance Computing

Performance

Visit CAPS Website:
www.caps-entreprise.com