Advanced Programming of ManyCore Systems
CAPS OpenACC Compilers

Stéphane BIHAN
NVIDIA GTC, March 20, 2013, San Jose, California
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

NVIDIA GTC, March 20, 2013, San Jose, California
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

- CEA
- AWE
- Oak Ridge National Laboratory
- Chang Gung Institute of Technology
- Total
- NORA
- TOHOKU UNIVERSITY

Resellers and Partners

- Main chip makers
- Hardware constructors
- Major software leaders

NVIDIA GTC, March 20, 2013, San Jose, California
Many-Core Technology Insights
Various Many-Core Paths

**AMD Discrete GPU**
- Large number of small cores
- Data parallelism is key
- PCIe to CPU connection

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

**INTEL Many Integrated Cores**
- 50+ number of x86 cores
- Support conventional programming
- Vectorization is key
- Run as an accelerator or standalone

**NVIDIA GPU**
- Large number of small cores
- Data parallelism is key
- Support nested and dynamic parallelism
- PCIe to host CPU or low power ARM CPU (CARMA)
Many-core Programming Models

Applications

Libraries
- Just use as is

Directives (OpenACC, OpenHMPP)
- Standard and high level

Programming languages (CUDA, OpenCL, ...)
- Maximum performance and maintenance cost

CAPS Compilers

Portability and Performance
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
  - Version 2 to be released

- 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

- OpenMP4 with extensions for accelerators
  - Release candidate 2 available
  - To be released in the coming months
Source to source compilers
  - Use your preferred native x86 and hardware vendor compilers
OpenACC and OpenHMPP
Directive-based Programming
Simply Accelerate with OpenACC

```c
#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 different kernel.
Each loop nest is launched in order in the device.

```c
#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] + ...;
      }
    }
  }
}
```

The entire *parallel* construct becomes a single kernel.

Add independent clause to disambiguate pointers.

Add the loop directives to distribute the work amongst the threads (redundant otherwise).
Gang, Workers and Vector in a Grid

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

```c
#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];
  }
}
}
```

Grid with ‘gang’ number of blocks

Block

Block with ‘worker’ number of threads

Vector: the number of inner loop iterations to be executed by a thread
OpenHMPP Codelet/Callsite Directives

- Accelerate function calls only

```c
#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**

```c
#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] = ...;
    }
}
```
What’s New In OpenACC 2.0

- Clarifications of some OpenACC 1.0 features
- Improved asynchronism
  - WAIT clause on PARALLEL, KERNELS, UPDATE and WAIT directives
  - ASYNC clause on WAIT so queues can wait for each others in a non-blocking way for the host
- ENTER DATA and EXIT DATA directives
  - Equivalent to the DATA directive but without syntactic constraints
- New TILE directive on ACC LOOP to decompose loops into tiles before worksharing
- Lot’s of new API calls
- Support for multiple types of devices
- ROUTINE directive for function calls within PARALLEL and KERNELS
- Simplification of the use of global data within routines
- Nested parallelism using parallel or kernel region containing another parallel or kernel region
OpenHMPP Advanced Programming
What’s in OpenHMPP not in OpenACC?

- Use specialized libraries with directives
- Zero-copy transfers on APUs
- Tuning directives
- Use native 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

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

NVIDIA GTC, March 20, 2013, San Jose, California
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

```c
#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
Integration of handwritten CUDA and OpenCL codes with directives

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

#include (<native.cu>)

#pragma hmppcg native (f1)
for( i = 0 ; i < size ; ++ )
    res[i] = f1( v1[i], v2[i], alpha, pow);
```

**Native function in CUDA**

```c
__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);
}
```

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

Kernel execution time

- Exploration phase
- Steady phase

Gang: 14
Worker: 16

Gang: 256
Worker: 128

NVIDIA GTC, March 20, 2013, San Jose, California
Dynamic Mapping of Kernels on a Grid

```c
size_t gangsVariant[] = { 14, 14, 14, 14, 14 , 14 , 14, 28, 28, 28, 28, 28, 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;
```
Gang and Vector Auto-tuning Results

NVIDIA Kepler

Performance (lower is better)

Run #

AMD Trinity APU

Performance (lower is better)

Run #

Gangs: 300
Vectors: 128

Gangs: 100
Vectors: 256

Gangs: 200
Vectors: 256

Gangs: 200
Vectors: 16

AMD Radeon

Performance (lower is better)

Run #

Intel MIC

Performance (lower is better)

Run #
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

### Workstation License

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Front-ends (C/Fortran/C++)</strong></td>
<td>C or Fortran</td>
<td>C, Fortran or C++</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td><strong>Programming Directives (OpenHMPP/OpenACC)</strong></td>
<td>OpenACC</td>
<td>All</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td><strong>Targets (CUDA/OpenCL)</strong></td>
<td>CUDA or OpenCL</td>
<td>CUDA or OpenCL</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td><strong>License Type</strong></td>
<td>User-based Node-locked</td>
<td>User-based Node-locked</td>
<td>User-based Node-locked</td>
<td>Floating</td>
</tr>
<tr>
<td><strong>Support</strong></td>
<td>Bronze</td>
<td>Silver</td>
<td>Gold</td>
<td>Gold</td>
</tr>
<tr>
<td><strong>Price</strong></td>
<td>199,00 €</td>
<td>399,00 €</td>
<td>999,00 €</td>
<td>2990,00 € (per token) 1290,00 € EDU</td>
</tr>
</tbody>
</table>

### Server License

- **Full Edition**
  - All

---

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


NVIDIA GTC, March 20, 2013, San Jose, California