Using CAPS Compiler on NVIDIA Kepler and CARMA Systems

F. Bodin
CTO – CAPS entreprise
Introduction

• CAPS develops programming tools to help writing a unique source code that can be executed on existing accelerator technologies
  o C / C++ / Fortran
• Fast moving hardware systems require two directives sets
  o OpenHMPP - easy to extend – integrate new HW features
  o OpenACC - standardized – longer term view but moving slowly
• Generates CUDA or OpenCL codes
  o Portable on AMD GPU and APU, Intel MIC, Nvidia Kepler-Carma, …
• Provide OpenACC and OpenHMPP directives
  o OpenHMPP codelet based
  o OpenACC code region based

```c
#pragma hmpp myfunc codelet, ...
void saxpy(int n, float alpha, float x[n], float y[n]){
    #pragma hmppcg gridify(i)
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
```

```c
#pragma acc kernels ...
{
    for(int i = 0; i<n; ++i)
        y[i] = alpha*x[i] + y[i];
}
```
Compilation Process

- Source-to-source technology
A Few Typical Situations

1. Simple nested loops
2. Data transfer optimization
3. Complex loop nests
4. Code tuning
5. Integrating auto-tuning techniques
6. Dealing with accelerated library
7. Dealing with dynamic accelerated tasks scheduling
8. Using multiple accelerators
9. Nested parallelism using native techniques
Simple nested loops - 1

- The simple construct is to declare a parallel loop to be compiled and executed on an accelerator
  - Iterations of the loop nests are converted into threads

- Data in and out declaration is used to determine the data to move between the host and the accelerator
Simple nested loops - 2

- Example of stencil computation

```c
#pragma acc kernels pcopyin(A[0:m]) pcopy(B[0:m])
{
  float c11, c12, c13, c21, c22, c23, c31, c32, c33;

  c11 = +2.0f; c21 = +5.0f; c31 = -8.0f; ...
  #pragma acc loop independent
  for (int i = 1; i < M - 1; ++i){
    #pragma acc loop independent
    for (int j = 1; j < N - 1; ++j){
    }
  }
  ...
```
Data transfer optimizations - 1

- Data transfers between the host CPU and the accelerator may very negatively impact on performance.

- A set of directives are provided to keep data on the accelerator beyond the execution of one kernel.
Data Transfer Optimization - 2

- Example from HydroC*

```c
void hydro_godunov (...) {
#pragma acc data
create(qleft[0:H.nvar], qright[0:H.nvar], ...
,...)

...\copy(uold[0:H.nvar*H.nxt*H.nyt]) \copyin(Hstep)
{
    for (j = Hmin; j < Hmax; j += Hstep){
        // compute many slices each pass
        int jend = j + Hstep;
        if (jend >= Hmax)
            jend = Hmax;
        . . .// the work here
    } // for j
} //end of data region
...```

Data are left on the GPU during the step loop. `pcopy` clauses are used into called routines

*Pierre-François Lavallée\textsuperscript{a}, Guillaume Colin de Verdière\textsuperscript{b}, Philippe Wautelet\textsuperscript{a}, Dimitri Lecas\textsuperscript{a}, Jean-Michel Dupays\textsuperscript{a}\textsuperscript{a}IDRIS/CNRS, \textsuperscript{b}CEA,Centre DAM
Non perfectly nested loops can be challenging to parallelize efficiently

- OpenACC parallel regions provide control over the parallelization scheme
- Requires to distribute the iteration spaces onto gangs and workers
Complex Loop Nests - 2
• Extract from NOAA Nonhydrostatic Icosahedral Model (NIM) code

```c
!$acc parallel present(nprox,prox,u,...) vector_length(1) num_workers(64) num_gangs(512)
!$acc loop gang private (rhsu,...) private(ipn,k,isn,...)
do ipn=ips,ipe
  n  = nprox(ipn)
  ipp1 = prox(1,ipn)
  ...
!$acc loop worker vector
  do k=1,nz-1
    rhsu(k,1) = cs(1,ipn)*u(k ,ipp1)...
    ...
  enddo !k-loop
  k=nz-1
  rhsu(k+1,1) = cs(1,ipn)*u(k ,ipp1)...
  ...
!$acc loop worker vector  private(wk)
do k=1,nz
  Lots of statements
  enddo !k-loop
!$acc loop seq
do isn = 1,nprox(ipn)
!$acc loop worker vector
do k=1,nz-1
  Tgtu(k,isn) = ...
  enddo !k-loop
  Tgtu(nz,isn) = 2.*Tgtu(nz-1,isn) - ...
end do  ! isn-loop
(continued from previous page)
!$acc loop seq
do isn = 1,nprox(ipn)
  isp=mod(isn,nprox(ipn))+1
!$acc loop worker vector
  do k = 2,nz-1
    ...
  end do ! k-loop
  sedgvar( 1,isn,ipn,1)=(zm(1,ipn)...
    ...
  end do  ! isn-loop
!$acc loop worker vector
  do k=1,nz
    kp1=min(nz,k+1)
    ...
  end do
  bedgvar(0,ipn,1)=...
enddo !ipn-loop
!$acc end parallel
(continued on next page)```
Accelerating Heat Transfer Ray-Tracing Code with CAPS OpenACC Compiler

- Speedup X 19
  - With a FERMI C2050
- Speedup X 34
  - With K20

**Performances for various numbers of rays and configurations.**

**PROMES Laboratory Application:**
ray-by-ray heat transfer simulation (DP)

**CAPS Experimentations** done on FERMI C2050 / K20 in comparison to Sandy Bridge E5-2687W

nvidia SC 2012

www.caps-entreprise.com
OpenACC on Nvidia CARMA

Accelerating Heat Transfer Ray-Tracing Code with CAPS OpenACC Compiler

- ARM CPU + Accelerator Target
- Speedup 12x ARM cores and CARMA GPU

CUDA on ARM
OpenHMPP MD Example

- Molecular dynamic codes (HLRS / Colin Glass)
- From the APOS project (http://apos-project.eu)

![Graph showing speedup vs. Nehalem @2.8GHz (Single Core)]

- GTX680 (CUDA)
- GTX680 (HMMP)
- Interlagos 48 cores (OpenMP)
- Sandy Bridge 32 threads (OpenMP)

source: HLRS
OpenHMPP Stereo Vision Example

- Stereo Matching (ESAW) on GTX 550 Ti
- Result from the ANR Compa project

From: Jinglin ZHANG, Jean-Francois NEZAN, Jean-Gabriel COUSIN, Erwan RAFFIN
"Implementation of Stereo Matching Using A High Level Compiler for Parallel Computing Acceleration"
IVCNZ ’12, November 26 - 28 2012, Dunedin, New Zealand
Code Tuning - 1

• The more optimized a code is, the less portable it is
  o Optimized code tends to saturate some hardware resources
  o Parallelism ROI varies a lot
    • i.e. # threads and workload need to be tuned
  o Many resources not virtualized on HWA (e.g. registers, #threads)

Example of an optimized versus a non optimized stencil code
Code Tuning - 2

• Express code transformations via directives

```c
#pragma hmpp <mygroup> sgemm codelet, args[tout].io=inout, &
#pragma hmpp &         args[*].mirror args[*].transfer=manual
void sgemm( float alphav[1], float betav[1], const float t1[SIZE][SIZE],
            const float t2[SIZE][SIZE], float tout[SIZE][SIZE] ) {

    int j, i;
    const float alpha = alphav[0], beta = betav[0];

    #pragma hmppcg(OCL) unroll i:4, j:4, split(i), noremainder(i,j), jam
    #pragma hmppcg gridify (j,i)
    for( j = 0 ; j < SIZE ; j++ ) {
        for( i = 0 ; i < SIZE ; i++ ) {
            int k;
            float prod = 0.0f;
            for( k = 0 ; k < SIZE ; k++ ) {
                prod += t1[k][i] * t2[j][k];
            }
            tout[j][i] = alpha * prod + beta * tout[j][i];
        }
    }
}
```

Loop transformations

Apply only when compiling for OpenCL
Integrating Auto-Tuning Techniques - 1

• Adaptation of the code @ runtime
  o Use multiple call to a kernel to find the most efficient one
• Need to create an optimization space to explore
  o Compiler issue → runtime configurable #gang, #worker, #vector
• Need a way to explore optimization space
  o Auto-tuning driver issue
  o may also focus on execution time or energy

Source code

HMPP Compiler

Autotunable executable code

hmpp profiling interface

auto-tuning driver

collect profiling data
explore the variants space
Integrating Auto-Tuning Techniques – 2-a

- Auto-tuning implementation of a Blur filter in OpenACC
- Explore dynamic parameters (e.g. #gangs, #workers)

```c
size_t gangs[] = { 8, 16, 32, 64, 128, 128, 8, 16, 32, 64, 128, 256 };
size_t workers[] = { 16, 16, 16, 16, 16, 16, 24, 24, 24, 24, 24, 24 };
...
while (nber_of_iterations < max_iterations) {
    ...
    variant = variantSelectorState("kernel.c:21",
        (sizeof(gangs)/sizeof(size_t))-1);
    blur(images[(currentImage + 1) % 2], image_caps, width, height,
        blockSize, gangs[variant], workers[variant]);
    ...
}
```
Integrating Auto-Tuning Techniques – 2-b

- Auto-tuning driver behavior with DNADist (bio info)

Kernel Computation Time (in sec). Lower is better

- Data can be collected over multiple executions
- kernel time
- exploration phase
- steady state

config. 8 = 14 G x 16 W
config 10 = 256 G x 128 W
Integrating Auto-Tuning Techniques - 3

- Variant based auto-tuning in HMPP
- Explore compile time code transformations / algorithms

```c
void filterStencil5x5_V2(const uint32 p_heigh[1],
const uint32 p_width[1],
const RasterType filter[5][5],
const RasterType *p_inRaster, RasterType *p_outRaster){
...
#pragma hmpp cg grid blocksize "32x4"
#pragma hmpp cg unroll 6, jam
for (i = stencil; i < heigh - stencil; i++) {
...
```

Variants of codelets to use: filterStencil5x5_V1 and filterStencil5x5_V2.
Dealing with Accelerated Library – 1

• Library calls can usually only be partially replaced
  o No one-to-one mapping between libraries (e.g. BLAS, FFTW, CuFFT, CULA, ArrayFire)
  o No access to all application codes (i.e. avoid side effects)
  o Want a unique source code

• Deal with multiple address spaces / multi-HWA
  o Data location may not be unique (copies, mirrors)
  o Usual library calls assume shared memory
  o Library efficiency depends on updated data location

• Libraries can be written in many different languages
  o CUDA, OpenCL, OpenHMPP, etc.
Using cuFFT accelerated version when source code is FFTW based
- A set of proxies implements the FFT accelerated version of the calls
- Allows resources sharing between the library and users' codes to reduce data transfers between host and accelerator memories
- Only marked calls are executed on the accelerators

```c
#pragma hmppalt cufft call, name="fftw_plan_dft_c2r_1d_sharing"
pc2r = fftw_plan_dft_c2r_1d(n, odata_intermediate,
      odata_real_GPU,FFTW_ESTIMATE);

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

#pragma hmpp <my_grp> filter callsite
filter(n, (double _Complex *)odata_intermediate, cf);

#pragma hmppalt cufft call, name="fftw_execute_sharing"
fftw_execute(pc2r);
```
Dealing with Dynamic Accelerated Tasks Scheduling – 1

- When dealing with a large bunch of small parallel tasks
  - Need to exploit accelerator asynchronous execution
- Madness (Multiresolution ADaptive NumErical Scientific Simulation)
  - Madness integrates its own tasks manager
  - Study in collaboration with ORNL

![Diagram showing task scheduling and resource allocation]

**Accelerator**

- Q1
- Q2
- Q3

**Task C/Fortran source code**

**OCL/CUDA code**

**CAPS compiler**

- T0,T6,T4,T5
- T1,T2,T3,T7
- T8,T9

 dependent tasks are sent to the same queue
Dealing with Dynamic Accelerated Tasks

Scheduling – 2

- Example of task queuing
- Exploit Caps compiler code generation
- Close to OpenCL API but task code remains C/Fortran one

for(int i=0; i<nb_arrays;i++){  
    hmpprt::Queue *myQueue = (hmpprt::Queue *) myQueues[i];
    myQueue->enqueueUpload(g_A[i],h_A[i]);
    myQueue->enqueueUpload(g_B[i],h_B[i]);

    hmpprt::ArgumentList myArguments;
    myArguments.addArgument(g_A[i]);
    myArguments.addArgument(g_B[i]);
    myArguments.addArgument(g_C[i]);

    myQueue->enqueueCall(myDevice, myCodelet, myArguments);
    myQueue->enqueueDownload(g_C[i],h_C[i]);
}

for(int i=0; i<nb_arrays;i++){  
    hmpprt::Queue *myQueue = (hmpprt::Queue *) myQueues[i];
    myQueue->start();
}

for(int i=0; i<nb_arrays;i++){  
    hmpprt::Queue *myQueue = (hmpprt::Queue *) myQueues[i];
    myQueue->wait();
}
Using Multiple Accelerators - 1

- Nodes can have multiple accelerators

- One thread / one MPI process mapping very limited

- Having to use CPU parallel code for exploiting multiple accelerator is inconvenient

- OpenHMPP multi-accelerator features based on
  - Data distribution
  - Owner compute rule to allocate the tasks/codelets to the device
Using Multiple Accelerators - 2

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

![Diagram showing the use of multiple GPUs with shared memory](image)
Using Multiple Accelerators - 3

- A parallel for loop implements a "map" operation

```c
#pragma hmpp parallel, device="i%2"
for( i=0;i<NB;i++){  
  //Allocate the mirrors for vin1, vin2 and vout
#pragma hmpp <mygroup> allocate, data["vin1[i]","vin2[i]", "vout_multi[i]"], size={size,size}, elementsize="4"
  ... 
}
#pragma hmpp parallel
for( i=0;i<NB;i++) {
  //launch the codelet on the device
#pragma hmpp <mygroup> sgemm callsite
  sgemm( vin1[i], vin2[i], vout_multi[i] );
}
```

- Distribute the data over the accelerators
- Execute the task in parallel according to the owner compute rule
Nested parallelism using native – 1

- Exploit K20 new capabilities for nested parallelism

```c
#pragma hmpp cudap codelet, target=CUDA, args[A].io=in, ...

void codelet(float *A, float *B, float *C, int nm, int ms, int size)
{
    int i;
    #pragma hmppcg gridify
    for(i = 0 ; i < 1 ; i++)
    {
        #pragma hmppcg(CUDA) include("native_dynamic_parallelism.h")
        #pragma hmppcg(CUDA) native(dynamic_parallelism)
        dynamic_parallelism(A, B, C, nm, ms, size);
    }
}

void dynamic_parallelism(float *A, float *B, float *C, ...)
{
    #ifndef __HMPP
    int i, j, k, l;
    for (k = 0 ; k < nm ; k++) {
        for (l = 0 ; l < nm ; l++) {
            ...
            cblas_sgemm(CblasColMajor, CblasNoTrans, CblasNoTrans, ms, ms, ms, 1.0f, loA, nm * ms, loB, nm * ms, 1.0f, loC, nm * ms);
        }
    }
    for (i = 0 ; i < ms ; i++) {
        for (j = 0 ; j < ms ; j++) {
            loC[access(i,j)] /= 2.f;
        }
    }
    ...
```
Nested parallelism using native – 2

• The native CUDA device function

```c
__device__ void dynamic_parallelism(float *A, float *B, float *C, ...) {
    cublasStatus_t status;
    cublasHandle_t handle;
    int i, j;
    dim3 g(ms/8,ms/8,1);
    dim3 b(8,8,1);
    status = cublasCreate(&handle);
    for (i = 0 ; i < nm ; i++) {
        for (j = 0 ; j < nm ; j++) {
            float loA = A + offset(i,j); float *loB = B + offset(i,j);
            float *loC = C + offset(i,j);
            const float alpha = 1.f; const float beta = 1.f;
            cudaStream_t s;
            cudaStreamCreateWithFlags(&s, cudaStreamNonBlocking);
            cublasSetStream(handle, s);
            cublasSgemm(handle, CUBLAS_OP_N, CUBLAS_OP_N, ms, ms, ms,...);
            halv<<<g,b,0,s>>>(loC,nm,ms);
            cudaStreamDestroy(s);
        }
    }
    cudaDeviceSynchronize();
    status = cublasDestroy(handle);
}
```
Conclusion

• CAPS technology provides a flexible and portable programming environment for accelerator based systems
  o OpenACC and OpenHMPP are complementary sets of directives
  o Many features to handle various coding issues
  o CPU/Accelerator library integration important for code maintenance

• Auto-tuning techniques helps to simplify code tuning and deployment
  o Code adapts to the architecture configuration

See a demo on Booth #2330