Using OpenMP to Program Embedded Heterogeneous Systems

Eric Stotzer, PhD
Senior Member Technical Staff
Software Development Organization, Compiler Team
Texas Instruments
February 16, 2012
Presented at SIAM-PP12 Savannah, GA
Abstract

Efficiency improvements with respect to power, performance, and silicon area motivate the specialization of processor resources including accelerators. Heterogeneous processors and accelerators are common in embedded computing, where they tend to be programmed using low-level vendor specific APIs. Currently, the dominant heterogeneous system in general purpose computing is the general purpose host CPU + GPU accelerator platform. The dominant programming models for these systems are currently CUDA and OpenCL. Both allow the programmer to extract performance from the accelerator, but the low level approach is repetitive, error-prone and focused on data movement. Compilers can implement significant portions of this repetitive code.

The OpenMP language committee is working on extensions to OpenMP that allow the programmer to accelerate key kernels or entire applications by adding directives to the original source code (Fortran, C or C++). The extensions support rapid, maintainable accelerator code development while leaving performance optimizations to the compiler and runtime environment. In keeping with the philosophy of OpenMP, these directives do not alter the existing code that runs well on the host CPU. In this presentation, we will present an overview of the current proposal for extending OpenMP to program accelerators and how it can also be used as a more general heterogeneous multicore programming model.
Outline

- Why OpenMP
- OpenMP in embedded systems
- OpenMP for accelerators (heterogeneous systems)
Why OpenMP?

• Industry standard for shared memory parallel programming
  – website: http://www.openmp.org

• Productivity and flexibility
  – Data parallelism (omp parallel for)
  – Task parallelism (omp tasks)

• Easy migration for existing code base: C/C++ based directives (#pragma) used to express parallelism

• Language is evolving to support tasking models, heterogeneous systems, and streaming programming models

• Embedded programmers want a standard parallel programming model for high performance shared memory multicore devices
OpenMP Execution Model

- **Master thread** spawns a team of threads as needed.
- Parallelism is added incrementally until desired performance is achieved: i.e. the sequential program evolves into a parallel program.
OpenMP Memory Model

• Threads have access to a *shared* memory
  – for *shared* data
  – each thread can have a temporary view of the shared memory
    (e.g. registers, cache, etc.) between synchronization barriers.

• Threads have *private* memory
  – for *private* data
  – Each thread has a stack
  – data local to each task it executes
OpenMP: Parallel Regions

• You create threads in OpenMP with the “omp parallel” pragma.
• For example, To create a 4-thread Parallel region:

```c
float A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int id = omp_get_thread_num();
    lots_of_work(id, A);
}
```

Each thread redundantly executes the code within the structured block.

Each thread calls lots_of_work(id,A) for id = 0 to 3
OpenMP: Work-sharing construct

```c
#pragma omp parallel for reduction(+:s)
for (int i=0; i<N; i++)
    s += x[i] * c[i];
```

- A single copy of x[] and c[] is shared by all the threads
OpenMP: Task Construct

- Task model supports irregular data dependent parallelism
- Tasks are assigned to a queue
- Threads execute tasks that they remove from a task queue

```c
#pragma omp parallel
{
    #pragma omp single
    {
        p = listhead;
        while (p) {
            #pragma omp task
            process(p);
            p = next(p);
        }
    }
}
```
C66x CorePac Overview

- 16 Flops/cycle (6 DP/cycle)
- 8-Wide VLIW Architecture
- Integrated fixed point / floating point
- 64-bit datapaths on .L and .S, 128-bit datapaths on .M unit
- Automated L2 cache Pre-fetch
- Integrated Power Switches
Embedded HPC

Mission Critical

High Performance Imaging

Medical Imaging

Software Defined Radio

High Performance Multimedia

Multichannel & Next Generation Video – H.265, SVC, etc.
Shannon (TMS320C6678) – Block Diagram

- Multi-Core KeyStone SoC
- Fixed/Floating CorePac
  - 8 CorePac @ 1.25 GHz
  - 4.0 MB Shared L2
  - 320G MAC, 160G FLOP, 60G DFLOPS
  - ~10W
- Navigator
  - Queue Manager, Packet DMA
- Multicore Shared Memory Controller
  - Low latency, high bandwidth memory access
- Network Coprocessor
  - IPv4/IPv6 Network interface solution
  - 1.5M pps (1Gb Ethernet wire-rate)
  - IPSec, SRTP, Encryption fully offloaded
- 3-port GigE Switch (Layer 2)
- HyperLink
  - 50G Baud Expansion Port
  - Transparent to Software
OpenMP on a lightweight RTOS

- TI SYS/BIOS RTOS with IPC product
- Each core is running SYS/BIOS RTOS
- OpenMP master and worker threads execute inside dedicated SYS/BIOS tasks
- IPC is used for communication and synchronization
- OpenMP run-time state and user data is allocated in shared memory
Quad-Shannon PCIe Development Card

- 512 Gflops
- 50 W

- Available from Advantech
- 8 Lanes PCIe out, sRIO / Hyperlink for intra-DSP data movement
- 8xShannon board under development
# Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>C6678</th>
<th>TI Quad PCIe</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP Flops</td>
<td>160</td>
<td>512</td>
</tr>
<tr>
<td>Cores</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>Processor speed</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>L2 Memory (MB)</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>L3 Memory (GB) ECC</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>Memory BW (GB/s)</td>
<td>12.8</td>
<td>42.6</td>
</tr>
<tr>
<td>Power Consumption</td>
<td>10</td>
<td>54</td>
</tr>
</tbody>
</table>

### Benchmarks (Gflops)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Gflops/W</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT (SP) (4096pt)</td>
<td>6.00</td>
</tr>
<tr>
<td>FFT Gflops/W</td>
<td>4.44</td>
</tr>
<tr>
<td>SGEMM (matrix multiply)</td>
<td>7.2</td>
</tr>
</tbody>
</table>

- **C6678 provides the highest performance/W**
- **Standard programming model**
OpenMP Accelerator programming model

- Why a new model? There are already many ways to program:
  - CUDA, OpenCL, MCA, vendor-specific
  - All are quite low-level and closely coupled to GPUs
  - Hard to write, debug, and port to new platforms

- Directives provide high-level approach
  - Based on original source code
  - Easier to maintain/port/extend code
  - Run the same code on a multi-core CPU

- What about OpenHMPP, OpenACC, PGI accelerator, etc...?
  - Unclear how these directives interact with existing OpenMP programs
  - OpenMP will borrow ideas from these approaches
  - gcc implements OpenMP 3.1

- OpenMP accelerator sub-committee
  - Members: Cray (co-chair), Texas Instruments (co-chair), Intel, Nvidia, PGI, IBM, CAPS, NASA, BSC, TACC, LLNL, Compunity, and more.
  - Goal: an integrated directive based approach that supports data, task and accelerator-style parallelism (not just gpu) available in the 4.0 OpenMP specification.
Disclaimer

• The OpenMP accelerator sub-committee is still working on the proposal.

• The names and syntax are subject to change.

• Your getting my (non-gpu) perspective.
Serial for-loop

```c
int main(int argc, char *argv[]) {
    int n = ...;
    float *x, *y;
    x = new float[n+1];
    y = new float[n+1];

    ... // fill x, y

    // do computation
    float e = 0;
    for (int i=1; i<n; ++i) {
        x[i] += ( y[i+1] + y[i-1] ) *.5;
        e += y[i] * y[i];
    }

    ... // output x, e

    delete[] x, y;
    return 0;
}
```

CUDA for-loop (1/2)

```c
#include <cuda.h>

// kernel
__global__ void sub1(float* fx, float* fy, float* fe) {
#define BLOCK (512)
  int t = threadIdx.x; // builtin
  int b = blockIdx.x; // builtin
  float e;

  __shared__ float se[BLOCK];
  __shared__ float sx[BLOCK];
  __shared__ float sy[BLOCK+2];
  // copy from device to processor memory
  sx[t] = fx[BLOCK*b+t];
  sy[t] = fy[BLOCK*b+t];
  if (t<2)
    sy[t+BLOCK] = fy[BLOCK*b+t+BLOCK];
  __syncthreads();

  // do computation
  sx[t] += (sy[t+2] + sy[t]) *.5;
  e = sy[t+1] * sy[t+1];
  // copy to device memory
  fx[BLOCK*b+t] = sx[t];
  // reduction
  se[t] = e;
  __syncthreads();
  if (t<256) {
    se[t] += se[t+256];
    __syncthreads();
  }
  if (t<128) {
    se[t] += se[t+128];
    __syncthreads();
  }
  if (t<64) {
    se[t] += se[t+64];
    __syncthreads();
  }
  if (t<32) { // warp size
    se[t] += se[t+32];
    se[t] += se[t+16];
    se[t] += se[t+8];
    se[t] += se[t+4];
    se[t] += se[t+2];
    se[t] += se[t+1];
  }
  if (t==0)
    fe[b] = se[0];

```
int main(int argc, char *argv[]) {
    int n = ...;
    float *x, *y;
    x = new float[n+1];
    y = new float[n+1];

    ... // fill x, y

    // allocate GPU memory
    float *fx, *fy, *fe;
    cudaMalloc((void**)&fx, (n-1+2) * sizeof(float));
    cudaMalloc((void**)&fy, (n-1+2) * sizeof(float));
    cudaMalloc((void**)&fe, (n-1+2)/BLOCK * sizeof(float));
    float *de = new float[(n-1+2)/BLOCK];
    // copy to GPU memory
    cudaMemcpy(fx+1, &x[1],
               (n-1) * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(fy, &y[1-1],
                (n-1+2) * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcp yDeviceToHost;
    cudaMemcpy(fe, &de[1-1], (n-1+2)/BLOCK * sizeof(float),
               cudaMemcpyDeviceToHost);
    // release GPU memory
    cudaFree(fe);
    cudaFree(fy);
    cudaFree(fx);
    float e_local = 0;
    for (int i=0; i<(n-1+2)/BLOCK; ++i)
        e_local += de[i];
    e += e_local;
    delete[] de;

    ... // output x, e
    delete[] x, y;
    return 0;
}

... // call GPU
sub1<<<dimGrid, dimBlock>>>(fx, fy, fe);
... // copy to host memory
cudaMemcpy(fx+1, &x[1], (n-1) * sizeof(float),
           cudaMemcpyDeviceToHost);
OpenCL for-loop (1/2)

```c
#include <cl.h>
#include <malloc.h>

// kernel
#define BLOCK (512)
const char *source = 
"__kernel void sub1(__global float* fx,
    __global const float* fy,
    __local float* se, __global float* fe) {
    const unsigned int t = get_global_id(0);
    const unsigned int b = get_group_id(0);
    const unsigned block = 512;
    const unsigned int i = block*b+t;
    float e;
    /* do computation */
    fx[t] += ( fy[t+2] + fy[t] )*0.5;
    e = fy[t+1] * fy[t+1];
    /* reduction */
    se[t] = e;
    barrier(CLK_LOCAL_MEM_FENCE);
    if (t<256) {
        se[t] += se[t+256];
        barrier(CLK_LOCAL_MEM_FENCE);
    }\n    if (t<128) {
        se[t] += se[t+128];
    }\n    if (t<64) {
        se[t] += se[t+64];
        barrier(CLK_LOCAL_MEM_FENCE);
    }\n    if (t<32) {
        se[t] += se[t+32];
        se[t] += se[t+16];
        se[t] += se[t+8];
        se[t] += se[t+4];
        se[t] += se[t+2];
        se[t] += se[t+1];
    }\n    if (t==0){
        fe[b] = se[0];
    }";
```

int main(int argc, char *argv[]) {
    int n = ...;
    float *x, *y;
    x = new float[n+1];
    y = new float[n+1];
    ... // fill x, y

    // allocate GPU memory
    cl_command_queue queue = clCreateCommandQueue(ct, 
        aDevices[0], 0, 0);
    // allocate GPU memory
    cl_mem fx = clCreateBuffer(ct, 
        CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 
        (n-1)*sizeof(cl_float), &x[1], 0);
    cl_mem fy = clCreateBuffer(ct, 
        CL_MEM_READ_ONLY, 
        (n-1) * sizeof(cl_float), &y[1], 0);
    cl_mem fe = clCreateBuffer(ct, 
        CL_MEM_WRITE_ONLY, 
        (n-1)/BLOCK*sizeof(cl_float), 0, 0);
    cl_mem se = clCreateBuffer(ct, 
        CL_MEM_READ_WRITE, 
        BLOCK*sizeof(cl_float), 0, 0);
    cl_device_id *aDevices = (cl_device_id*)malloc(ctsize);
    clGetContextInfo(ct, 
        CL_CONTEXT_DEVICES, 0, 0, &ctsize);
    clGetContextInfo(ct, 
        CL_CONTEXT_DEVICES, ctsize, 
        aDevices, 0);
    // compile kernel
    cl_program prog = clCreateProgramWithSource(ct, 1, 
        &source, 0, 0);
    clBuildProgram(prog, 0, 0, 0, 0, 0);
    cl_kernel kern = clCreateKernel(prog, 
        "sub1", 0);
    float e = 0;
    // call GPU
    cl_kernel kern = clCreateKernel(prog, 
        "sub1", 0);
    float e = 0;
    cl_command_queue queue = 
        clCreateCommandQueue(ct, 
            aDevices[0], 0, 0);
    // allocate GPU memory
    cl_mem fx = clCreateBuffer(ct, 
        CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 
        (n-1)*sizeof(cl_float), &x[1], 0);
    cl_mem fy = clCreateBuffer(ct, 
        CL_MEM_READ_ONLY | 
        CL_MEM_COPY_HOST_PTR, 
        (n-1) * sizeof(cl_float), &x[1], 0);
    cl_mem fe = clCreateBuffer(ct, 
        CL_MEM_WRITE_ONLY, 
        (n-1)/BLOCK*sizeof(cl_float), 0, 0);
    cl_mem se = clCreateBuffer(ct, 
        CL_MEM_READ_WRITE, 
        BLOCK*sizeof(cl_float), 0, 0);
    cl_device_id *aDevices = (cl_device_id*)malloc(ctsize);
    clGetContextInfo(ct, 
        CL_CONTEXT_DEVICES, 0, 0, &ctsize);
    clGetContextInfo(ct, 
        CL_CONTEXT_DEVICES, ctsize, 
        aDevices, 0);
    // compile kernel
    cl_program prog = clCreateProgramWithSource(ct, 1, 
        &source, 0, 0);
    clBuildProgram(prog, 0, 0, 0, 0, 0);
    cl_kernel kern = clCreateKernel(prog, 
        "sub1", 0);
    float e = 0;
    cl_command_queue queue = 
        clCreateCommandQueue(ct, 
            aDevices[0], 0, 0);
    // allocate GPU memory
    cl_mem fx = clCreateBuffer(ct, 
        CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 
        (n-1)*sizeof(cl_float), &x[1], 0);
    cl_mem fy = clCreateBuffer(ct, 
        CL_MEM_READ_ONLY | 
        CL_MEM_COPY_HOST_PTR, 
        (n-1) * sizeof(cl_float), &x[1], 0);
    cl_mem fe = clCreateBuffer(ct, 
        CL_MEM_WRITE_ONLY, 
        (n-1)/BLOCK*sizeof(cl_float), 0, 0);
    cl_mem se = clCreateBuffer(ct, 
        CL_MEM_READ_WRITE, 
        BLOCK*sizeof(cl_float), 0, 0);
    cl_device_id *aDevices = (cl_device_id*)malloc(ctsize);
    clGetContextInfo(ct, 
        CL_CONTEXT_DEVICES, 0, 0, &ctsize);
    clGetContextInfo(ct, 
        CL_CONTEXT_DEVICES, ctsize, 
        aDevices, 0);
    // compile kernel
    cl_program prog = clCreateProgramWithSource(ct, 1, 
        &source, 0, 0);
    clBuildProgram(prog, 0, 0, 0, 0, 0);
    cl_kernel kern = clCreateKernel(prog, 
        "sub1", 0);
    float e = 0;
    cl_command_queue queue = 
        clCreateCommandQueue(ct, 
            aDevices[0], 0, 0);
    // allocate GPU memory
    cl_mem fx = clCreateBuffer(ct, 
        CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, 
        (n-1)*sizeof(cl_float), &x[1], 0);
    cl_mem fy = clCreateBuffer(ct, 
        CL_MEM_READ_ONLY | 
        CL_MEM_COPY_HOST_PTR, 
        (n-1) * sizeof(cl_float), &x[1], 0);
    cl_mem fe = clCreateBuffer(ct, 
        CL_MEM_WRITE_ONLY, 
        (n-1)/BLOCK*sizeof(cl_float), 0, 0);
    float e = 0;
    for (int i=1; i<n; ++i) { 
        x[i] += ( y[i+1] + y[i-1] )*.5;e += y[i] * y[i];
    } 
    // release GPU memory
    clReleaseMemObject(fx);
    clReleaseMemObject(fy);
    clReleaseMemObject(se);
    ... // output x, e
    delete[] d;
    return 0;
}
accelerate construct

• Goal: Extend OpenMP to support accelerator parallelism
• Model: “Offload” data and code to an accelerator.
• Accelerate region: A region of code that executes using accelerator parallelism.

```c
... // fill x, y

// do computation
float e = 0;
#pragma omp accelerate loop reduction(e:+) local(e(dsp))
for (int i=1; i<n; ++i) {
    x[i] += ( y[i+1] + y[i-1] )*.5;
    e += y[i] * y[i];
}

... // output x, e
```
Execution and Memory Model

- An accelerate region “escapes” the OpenMP thread based execution model
- Optimize memory transfers to realize performance entitlement
- Memory consistency: makes it “easy” and legal for a compiler to optimize away transfer operations (memory might actually be shared).
data region construct

• Define a data environment
  – Keep data persistent on accelerator to minimize memory transfers

• Data region spans two accelerator regions
  ▪ The runtime checks if b is already on the accelerator:
    ▪ yes: it uses this without copies; no: it follows the copy(b) clause
    ▪ Can also call double_me() from outside a data region

• Do not need to inline the subroutine (manually or by compiler)
  ▪ Can even be in different source file

```fortran
PROGRAM main
  REAL :: a(N)
  !$omp data alloc(a)
  !$omp accelerate loop
    DO i = 1,N
      a(i) = i
    ENDDO
  !$omp end accelerate loop
  CALL double_me(a)
END PROGRAM main

SUBROUTINE double_me(b)
  REAL :: b(N)
  !$omp accelerate loop present(b) copy(b)
    DO i = 1,N
      b(i) = 2*b(i)
    ENDDO
END SUBROUTINE double_me
```
Loop Construct - example

- We cannot describe accelerator-style parallelism using OpenMP constructs
  - i.e. GPU warps/blocks/kernels/grids
  - simd vector size
- Need more control expressing non-thread-style parallelism to exploit accelerator hardware
- In this example think of the simd datapath as an accelerator

```c
#define NLANES 4     // width of SIMD datapath

void add2d(char A[], char B[], char C[], int nrows, int ncols)
{
    #pragma omp parallel for schedule(static)
    for (int i=0; i < nrows; i++)          // go parallel on threads
    {
        #pragma omp accelerate loop simd(NLANES) locale(SSE)
        for (int j=0; j < ncols; j++)      // each thread uses SIMD
        {
            int index = (i * ncols) + j;
        }
    }
}
```
F28M3x Concerto™ Series

**Control Subsystem**
- Precision Control
  - Industry leading computational performance
  - Expanded instruction set
  - Industry’s highest-resolution PWMs
- Low-latency control loops
- Real-world, modular control software
- High-speed precision analog
- Fine-tuned control architecture

**Host Subsystem**
- Ecosystem for Developers
  - Operating System
  - Middleware
  - SW Infrastructure
- Robust Communications
  - Ethernet
  - CAN
  - Fieldbus
  - USB
- Additional functions
  - Natural user interface
  - Motion profile
  - Safety

---

**Control Subsystem**

- **C28x 32-bit CPU**
  - Up to 150 MHz
- **VCU**
  - Viterbi
  - CRC
  - Complex MPY
  - FFT
- **Comms**
  - McBSP/SP/12S
  - UART

**System**
- 6Ch DMA

**Parity RAM**
- 2 KB Message
- 2 KB Message
- Up to 64 KB

**Pwr & Clocking**
- 10 MHz / 30 KHz INT GSC
- 40 MHz EXT
- Clock Fail Detect
- 3.3V VREG
- POR/BOR

**Shared**

- **Analog Temp Sense**
  - 12b, 10ch, 2SH, 3 MSPS
  - 3ch Analog Comparator
- **ARM® Cortex™-M3**
  - 32-bit CPU
  - Up to 100 MHz

**Host Subsystem**

- **Memory**
  - 256-512 KB ECC Flash
  - 20 KB ECC RAM
  - 128-bit Security
  - 16 KB Parity RAM
  - 64 KB ROM
- **Communications**
  - 10/100 Ethernet MAC
  - 1588 w/ MII
  - USB OTG FS PHY
  - 4x SSI
  - 5x UART
  - 2x PC
  - 2x CAN

---

**System & Clocking**
- 32Ch DMA
- 4 Timers
- 2 Watchdogs
DSP HPC Cluster Architecture

- Compute Nodes (Shannon)
- I/O Nodes (ARM A15 or other) running Linux
- Linux-compatible Compute Node Kernel
- High speed interconnect
Heterogeneous dispatch construct

- Dispatch (offload) code/data to a place (locale)
- Creates a new OpenMP environment

```c
// Code executing on arm
f();
#pragma omp dispatch copyin(A,B) copyout(C) locale(dsp0)
{
    // New OpenMP environment on dsp0

    #pragma omp parallel for schedule(static)
    for (int i=0; i < nrows; i++) // go parallel on threads
    {
        #pragma omp accelerate loop simd(NLANES)
        for (int j=0; j < ncols; j++) // each thread uses SIMD
        {
            int index = (i * ncols) + j;
        }
    }
}
#pragma omp taskwait // wait for dispatched region to complete
// more code executing on arm
g();
```
Summary

- OpenMP is the industry standard for shared memory parallel programming (or directive based parallel programming?)

- OpenMP can execute on an embedded RTOS or perhaps even “bare-metal”

- OpenMP will be successful in embedded systems:
  - Just like other high level languages have been adapted to embedded systems

- HPC computing looks like embedded computing
  - Performance and Power

- Accelerator(s) (heterogeneous) programming model
  - Accelerator regions: use accelerator parallelism (power-tool for a thread)
  - Data regions: Copy (Stream via DMA) data in/out from host to accelerator
  - Dispatch regions: more general heterogeneous offload capability