Generic Development Tools for Many-Core and Heterogeneous Processors

WinnForum Webinar – February 2012
1. Background
2. Heterogeneous and Many-Core Processors
3. Low-Level Software Development Tools
4. Generic Software Development Tools
5. Case study: Cognitive Network Simulation
1. Background
   • Top 10 Most Wanted Wireless Innovations
   • Motivation

2. Heterogeneous and Many-Core Processors
3. Low-Level Software Development Tools
4. Generic Software Development Tools
5. Case study: Cognitive Network Simulation
Top 10 Most Wanted Wireless Innovations

1. Techniques for Efficient Software Porting Between Heterogeneous Platforms
2. Generic Development Tools for Heterogeneous Processors
3. Certification Process for Third Party Waveform Software
4. Low Cost Wide Spectral Range RF Front-End (Multi-octave Contiguous) (Tx,Rx)
   • ...

October 2011)
Motivation

• Portable code across different platforms

• Standardized and integrated process from design to implementation

• Eliminate the risk of software errors when transitioning from one tool to another

• Focus on the algorithms and the problem at hand rather than implementation details

→ Reduce the development cost and time to market
2. Heterogeneous and Many-Core Processors

1. Background

2. Heterogeneous and Many-Core Processors
   • Overview
   • Why Many-Core Processors?
   • Multi-core CPUs and GPUs
   • Applications in Wireless Communications

3. Low-Level Software Development Tools

4. Generic Software Development Tools

5. Case study: Cognitive Network Simulation
Overview

• Embedded
  • DSPs
  • FPGAs
  • CPUs
  • Heterogeneous: combinations of the above

• Servers and workstations
  • Multi-core CPUs (e.g. Intel, AMD, IBM)
  • GPGPUs (e.g. Nvidia, AMD) (heterogeneous, CPU is needed)
  • Heterogeneous: CPU and GPU on chip
Why Many-Core Processors?

- Higher Performance
- Higher Power Efficiency
- Lower cost

→ Compute-intensive problems
Multi-core CPUs and GPUs

- Presentation focuses only on workstation/servers processors

<table>
<thead>
<tr>
<th>Intel® Core™ i7-3960X</th>
<th>Nvidia Tesla M2090</th>
</tr>
</thead>
<tbody>
<tr>
<td>• 6 Cores</td>
<td>• 512 CUDA Cores</td>
</tr>
<tr>
<td>• 12 threads</td>
<td>• Hundred 1000s of threads</td>
</tr>
<tr>
<td>• ~70 GFLOPS (DP)</td>
<td>• ~665 GFLOPS (DP)</td>
</tr>
</tbody>
</table>

(Source: http://www.brightsideofnews.com/print/2011/11/14/review-intel-core-i7-3960x-and-intel-x79-dx79si.aspx)

Applications in Wireless Comm.

- Network modelling, planning and optimisation
- Base station positioning, frequency planning, transmit power control

http://www.awe-communications.com/
Applications in Wireless Comm.

- Real-time signal detection and analysis
- Spectral analysis, cyclostationary analysis, etc.
Applications in Wireless Comm.

- MIMO processing
- Sphere decoding, Probabilistic Data Association etc.

3. Low-Level Software Development Tools

1. Background
2. Heterogeneous and Many-Core Processors
3. Low-Level Software Development Tools
   • OpenMP
   • Sample: Monte-Carlo Pi
   • CUDA
   • Programming Challenge
4. Generic Software Development Tools
5. Case study: Cognitive Network Simulation
OpenMP Overview

• De-facto standard API for writing shared memory parallel applications in C, C++, and Fortran for CPUs

• Consists of:
  • Compiler directives
  • Run time routines
  • Environment variables

• Specification maintained by the OpenMP Architecture Review Board

• Need to explicitly address parallelism ➔ developers need to think parallel

• Code is not portable: supported on multi-core CPUs only

➔ Error prone
➔ Require expert parallel programming skills
Example: Monte-Carlo Pi

- Monte-Carlo algorithm for estimating the value of Pi
- Draw a square on the ground
- Draw a circle within it
- Scatter grains of rice over the square
- Count the number of grains within the circle
- $\pi/4 = \text{grains in circle/ total grains}$

$$\pi = 4 \frac{N_{\text{in\_circle}}}{N_{\text{total}}}$$
Example: Monte-Carlo Pi

- Flow graph for the algorithm:

\[ \pi = 4 \times \frac{\text{NumInCircle}}{\text{Total}} \]
... // includes
#define N 1000000

float uniform_rand(unsigned* seed, float lower, float upper) { ... }

int main ()
{
    int num_in_circ = 0;
    float x, y, dist;

    #pragma omp parallel
    {
        // seed random number generator for every thread
        unsigned rseed = initseed + omp_get_thread_num();
        #pragma omp for private(x,y,dist) reduction(+:num_in_circ) schedule(static)
        for (int i = 0; i < N; i++)
        {
            // generate uniform random numbers in 0..1
            x = uniform_rand(&rseed, 0.0f, 1.0f);
            y = uniform_rand(&rseed, 0.0f, 1.0f);
            dist = sqrtf(x*x + y*y);
            if ( dist < 1.0f )
                num_in_circ++;
        }
    }

    float pi = 4.0f * (float)num_in_circ / N;
    cout << " Value of Pi = " << pi << endl ;
}
CUDA Overview

- Nvidia standard for GPGPU programming
- Uses a dialect of C with extensions and restrictions
- Need to explicitly handle CPU ↔ GPU memory transfers
- Only supported on GPUs from Nvidia – code is not portable to multi-core CPUs
- Need to address many hardware details (e.g. threads, grids, blocks, memory hierarchy, synchronisation)

- Error prone
- Require expert parallel programming skills
... //includes
#define NUM_THREAD 512
#define NUM_BLOCK 65534

// Function to sum an array
__global__ void reduce0(float *g_odata) {
    __shared__ int sdata[NUM_THREADS];
    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_odata[i];
    __syncthreads();
    // do reduction in shared mem
    for (unsigned int s=1; s < blockDim.x; s *= 2) { // step = s x 2
        if (tid % (2*s) == 0) { // only threadIDs divisible by the step participate
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }
    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}

// main kernel
__global__ void monteCarlo(float *g_odata, int trials, curandState *states) {
    // unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int k, incircle;
    float x, y, z;
    incircle = 0;
    curand_init(1234, i, 0, &states[i]);
    for(k = 0; k < trials; k++){
        x = curand_uniform(&states[i]);
        y = curand_uniform(&states[i]);
        z = sqrt(x*x + y*y);
        if (z <= 1) incircle++;
    }
    __syncthreads();
    g_odata[i] = incircle;
}

// main prog
int main() {
    float* solution = (float*)calloc(100, sizeof(float));
    float *sumDev, *sumHost = (float*)calloc(NUM_BLOCK*NUM_THREAD, sizeof(float));
    int trials, total;
    curandState *devStates;
    trials = 100;
    total = trials*NUM_THREAD*NUM_BLOCK;
    dim3 dimGrid(NUM_BLOCK,1,1); // Grid dimensions
    dim3 dimBlock(NUM_THREAD,1,1); // Block dimensions
    size_t size = NUM_BLOCK*NUM_THREAD*sizeof(float); //Array memory size
    cudaMalloc((void **) &sumDev, size); // Allocate array on device
    cudaMalloc((void **) &devStates, NUM_BLOCK*NUM_THREADS*sizeof(curandState));
    // Do calculation on device by calling CUDA kernel
    monteCarlo <<<dimGrid, dimBlock>>> (sumDev, trials, devStates);
    // call reduction function to sum
    reduce0 <<<dimGrid, dimBlock>>> (sumDev);
    // Retrieve result from device and store it in host array
    cudaMemcpy(sumHost, sumDev, size, cudaMemcpyDeviceToHost);
    *solution = 4*(sumHost[0]/total);
    printf("%.4f\n", 1000, *solution);
    free (solution);
    return 0;
}
Great hardware potential

<table>
<thead>
<tr>
<th>Intel® Core™ i7-3960X</th>
<th>Nvidia Tesla M2090</th>
</tr>
</thead>
<tbody>
<tr>
<td>6 Cores</td>
<td>512 CUDA Cores</td>
</tr>
<tr>
<td>12 threads</td>
<td>Hundred1000s of threads</td>
</tr>
<tr>
<td>70 GFLOPS (DP)</td>
<td>665 GFLOPS (DP)</td>
</tr>
</tbody>
</table>

(Source: Linpack benchmark)


Problem is programmer productivity

- Understand underlying architectures
- Explicitly address parallelism (e.g. OpenMP, CUDA, SIMD)
- Learn specific languages (e.g. CUDA)
- Platform-specific source code (not portable)
- Error-prone programming (deadlocks, race conditions)
- Difficult debugging of parallel code
Generic Software Development Tools

1. Background
2. Heterogeneous and Many-Core Processors
3. Low-Level Software Development Tools
4. Xcelerit SDK
   • Overview
   • Architecture
   • Programming Model
   • Programming Interface
   • Code sample
   • Under the hood

5. Case study: Cognitive Network Simulation
Xcelerit SDK Overview

- http://www.xcelerit.com/xcelerit-sdk
- Toolkit to boost application performance
- Programming interface: C/C++
- Increase programmer productivity
- Automatic parallelisation
- Easy code refactoring
- Cross-platform/Generic
  - Hardware: CPUs, GPUs, clusters
  - OS: Windows, Linux
  - Compilers: VS 2008/2010, GCC 4.x, Intel 12.x
Xcelerit SDK Architecture

Development Tools
- IDE plugin
- Debugger
- Profiler

Library:
- Math functions
- Common actors

Under the hood

Source-to-Source Compiler
- Dataflow analysis
- Code safety checks
- Graph scheduling

Code Generators:
- OpenCL
- CUDA

- Actor fusion
- Parallelisation
- Memory optimisation

Code Generators:
- Multi-threaded CPU
- Cluster (MPI)

Parallel Source Code

Standard Compilers
(OpenCL, CUDA, C/C++)

Application Binary

- CPUs
  (Intel, AMD, IBM)

- GPUs
  (Nvidia, AMD)

- Cluster
  (many CPUs and GPUs)

Runtime Library
- Environment discovery
- Scheduling
- JIT compilation

- Safety checking
- Execution manager
- Built-in profiler
- Parallel logging
actor  has inputs and outputs, processes streams of inputs
source  provides data streams for graph
sink    consumes final outputs
Xcelerit SDK Programming Interface: Actors

- Actor interface: ports, parameters, and constant arrays:

  ![Diagram of actor interface]

  - Input / Output ports
  - Parameters to tune behaviour
  - Constant arrays: lookup tables
Actor Example: Adder

- Inputs and output are size 1 (only needs one item from each input to compute 1 output)
- Core algorithm accesses items using array notation
Xcelerit SDK Programming Interface: Sources and Sinks

- Sources: outputs only
- Sinks: inputs only

Workflow:
- **Derive from class** `Source`/`Sink`
- Add & register ports
- Set port sizes
- Add `run` method

```java
class MySource : public Source {
    // ...
}
class MySink : public Sink {
    // ...
}

Output<float> out1, out2;
Input<int> in;

out1.setProductionRate(123);
in.setConsumptionRate(42);

void run()
{
    /* produce/consume data */
}
```
Under the Hood: Automatic Parallelism Extraction

Pipeline Parallelism

Data parallelism (incl. SIMD)

Parallel Reductions

Cluster: data parallelism
Sequential C++ (Traditional)

```c++
// returns 1 if point (x, y) is in unit circle
int is_in_circle(float x, float y) {
    float dist = sqrt(x*x + y*y);
    if (dist <= 1.0f)
        return 1;
    else
        return 0;
}
```

```c++
... // includes
# define N 1000000

float uniform_rand(float lower, float upper) {
    // ... }

int main () {
    int num_in_circ = 0;
    for (int i = 0; i < N; i++)
    {
        // generate uniform random numbers in 0..1
        float randx = uniform_rand(0.0f, 1.0f);
        float randy = uniform_rand(0.0f, 1.0f);

        // add if point is in circle
        num_in_circ += is_in_circle(randx, randy);
    }

    float pi = 4.0f * (float)num_in_circ / N;
    cout << "Value of Pi = " << pi << endl ;
}
```

Xcelerit SDK

Xcelerit SDK: Monte-Carlo Pi

Xcelerit SDK

```c++
actor IsInCircle { // define interface
    __input float x[1], y[1];
    __output float out[1];
};

// do work : compute output value from inputs
actor_run<IsInCircle>() {
    float dist = sqrt(x[0]*x[0] + y[0]*y[0]);
    if (dist <= 1.0f) out[0] = 1;
    else out[0] = 0;
}
```

```c++
... // includes
# define N 1000000

int main () {
    xceleritSdk sdk; // initialise the SDK
    int num_in_circ = 0;

    // provided source : generates N random numbers
    UniformRand<float> randx(0.0f, 1.0f, N),
                           randy(0.0f, 1.0f, N);
    IsInCircle is_in_circle; // actor
    // provided sink : sum input data
    SumReduce<int> sum(&num_in_circ );

    // connect dataflow graph
    Flowgraph f;
    f += randx >> is_in_circle.x ,
        randy >> is_in_circle.y ,
        is_in_circle.out >> sum ;
    f.run();

    // run graph
    float pi = 4.0f * (float)num_in_circ / N;
    cout << "Value of Pi = " << pi << endl ;
}
```
• Overhead reduction (actor fusion)
• Error checking: memory out-of-bound, type safety, etc.
• Memory access optimisations
Xcelerit SDK Under the hood: Runtime Library

- Discover environment (CPUs, GPUs, machines, …)
- Schedule computation to CPUs or GPUs
- Select best implementation for target hardware
- Dynamic compilation to give best result on available hardware at runtime
- Manage and monitor execution
Benefit of a Dataflow-Based Approach

- Intuitive way of thinking for mathematicians, scientists in many domains
- Application-centric - No parallel constructs
- Completely abstract the hardware - No need to know what runs on what
- Other hardware like FPGAs, DSPs, can be supported in future with same source code
- Automatic optimisation: parallelisation, memory, ...
- Handles large volumes of data
- Supports many types of algorithms
5. Case study: Cognitive Network Simulation

1. Background
2. Heterogeneous and Many-Core Processors
3. Low-Level Software Development tools
4. Generic Software Development tools
5. Case study: Cognitive Network Simulation
Network Simulation Using Xcelerit SDK

- Game Theory simulation for cognitive radio network
- Test machine-learning algorithm for channel allocation on wireless network nodes
- Hundreds of thousands of simulations:
  - All possible topologies with N= 3, 4, ..., 7 nodes
  - Monte-Carlo with 100,000 paths for each topology
  - Each path: Game Theory simulation, testing if learning algorithm converges

<table>
<thead>
<tr>
<th>Nodes</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topologies</td>
<td>2</td>
<td>7</td>
<td>23</td>
<td>122</td>
<td>888</td>
</tr>
</tbody>
</table>

All topologies with 4 nodes.
Sequential C++ (Traditional)

```c
... // includes, defines, functions, structs, etc.
void learningAutomaton(randseed seed,
    const unsigned char graph[],
    float optPolicy[],
    long* numIter ) {
    ... // generate random numbers (using seed), iterate
    ... // over nodes / learning steps, break if converged

    // assign policy output
    for (int nc =0; nc < (NCHANNELS+1); nc++)
        for (int nn =0; nn < NNODES ; nn++)
            optPolicy[nc * NNODES + nn] = ...;
}
```

```cpp
... // defines and includes
class GraphReader { ... }; // read graph from file
class Writer { ... }; // write outputs to file
randseed randSeed () { ... } // generate random seeds
int main () {
    GraphReader reader ("graphs.dat");
    Writer writer("results.dat");
    unsigned char graph[NNODES * NNODES];
    randseed seed ;
    float optPolicy[(NCHANNELS+1)* NNODES];
    long numIterations ;
    while (reader.readNext(graph))
        for (int j = 0; j < NUM_GR; j++)
            seed = randSeed();
            learningAutomaton(seed, graph, optPolicy, &numIterations);
            writer.write(optPolicy, numIterations);
}
```

Xcelerit SDK

```c
... // includes, defines, functions, structs, etc.
actor LearningAutomaton { // interface definition
    __input randseed seed;[4];
    __input unsigned char graph[NNODES * NNODES];
    __output float optPolicy[(NCHANNELS+1)* NNODES];
    __output long numIter[1];
};
actor_run<LearningAutomaton>() { // core function
    ... // generate random numbers (using seed), iterate
    ... // over nodes / learning steps, break if converged

    // assign policy output
    for (int nc =0; nc < (NCHANNELS+1); nc++)
        for (int nn =0; nn < NNODES ; nn++)
            optPolicy[nc * NNODES + nn] = ...;
}
```

```cpp
... // defines and includes
class GraphReader : public Source { ... };
class Writer : public Sink { ... };
randseed randSeed () { ... }
int main () {
    xceleritSdk sdk; // initialise SDK

    // instantiate sources, sinks and actors
    GraphReader reader("graphs.dat");
    SourceFunctor<randseed> seedGen(&randSeed);
    LearningAutomaton automat;
    Writer writer("results.dat");

    // connect dataflow graph
    Flowgraph f ;
    f += reader >> automat.graph,
        seedGen >> automat.seed,
        automat.optPolicy >> writer.optPolicy,
        automat.numIter >> writer.numIter ;
    f.run (); // run flow graph
}
```
Xcelerit SDK Network Simulation: Performance

- **Test System:**
  - Dual Xeon E5620 CPU (16 hardware threads)
  - Dual Nvidia Tesla M2050 GPU
  - 24GB RAM
  - RedHat Enterprise Linux 5 (64bit)

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Topologies</th>
<th>Sequential</th>
<th>Xcelerit CPU</th>
<th>Xcelerit GPU</th>
<th>Speedup CPU</th>
<th>Speedup GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>00:01:40</td>
<td>00:00:06</td>
<td>00:00:01</td>
<td>15.9</td>
<td>141.8</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>00:08:37</td>
<td>00:00:37</td>
<td>00:00:04</td>
<td>14.1</td>
<td>147.2</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>00:36:27</td>
<td>00:02:41</td>
<td>00:00:16</td>
<td>13.6</td>
<td>133.5</td>
</tr>
<tr>
<td>6</td>
<td>122</td>
<td>03:56:48</td>
<td>00:17:49</td>
<td>00:01:45</td>
<td>13.3</td>
<td>135.0</td>
</tr>
<tr>
<td>7</td>
<td>888</td>
<td>34:32:00</td>
<td>02:20:56</td>
<td>00:14:33</td>
<td>14.7</td>
<td>142.4</td>
</tr>
</tbody>
</table>

34.5 hours down to 14.5 minutes with the Xcelerit SDK
Summary

- Multi-core CPUs and GPUs: great hardware potential for compute-intensive applications
- The low-level tools available to program them are not portable, and require expert parallel programming skills
- Reviewed a generic software development tools for multi-core CPUs and GPUs
- Showed the performance grain using such approach for a wireless simulation
  - Dramatic performance increase
  - Preserve programmer productivity
  - Portable, generic solution
Email: lahlou (dot) hicham (at) gmail (dot) com
LinkedIn: http://www.linkedin.com/in/hichamlahlou