



# **Generic Development Tools for Many-Core and Heterogeneous Processors**

WinnForum Webinar – February 2012

# Outline

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

## 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)**
- ...

(Source: Wireless Innovation Forum: <http://www.wirelessinnovation.org>  
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

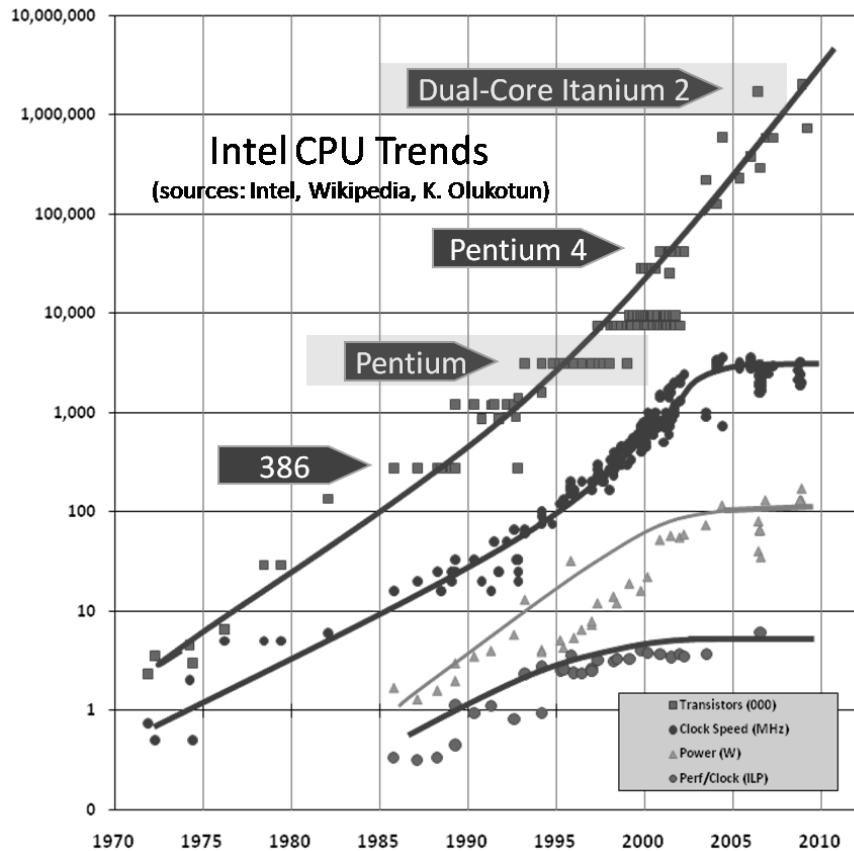
## **2. Heterogeneous and Many-Core Processors**

- 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

## Intel® Core™ i7-3960X

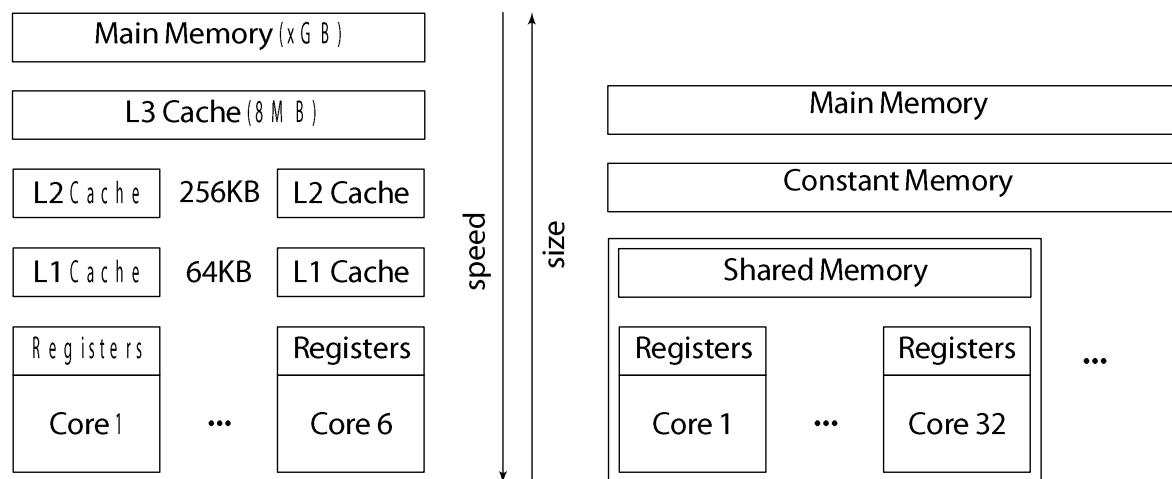
- 6 Cores
- 12 threads
- ~70 GFLOPS (DP)

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

## Nvidia Tesla M2090

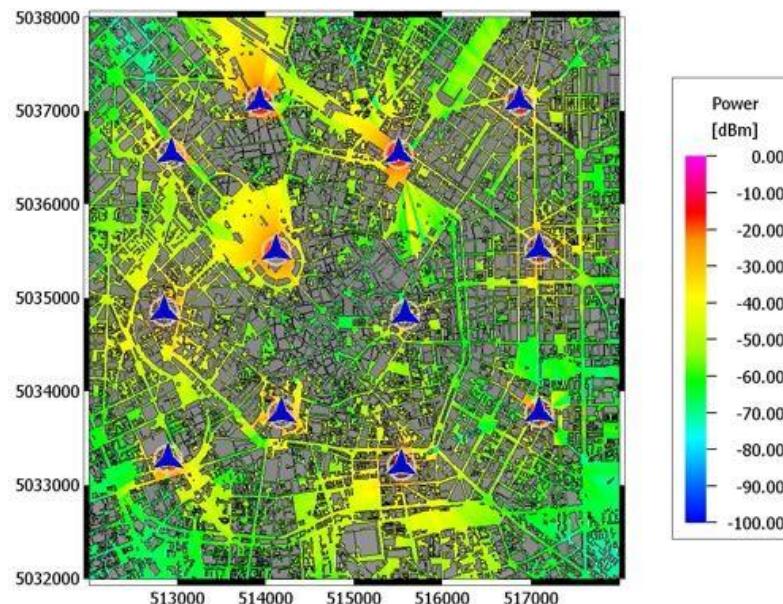
- 512 CUDA Cores
- Hundred 1000s of threads
- ~665 GFLOPS (DP)

(source: <http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf>)



# Applications in Wireless Comm.

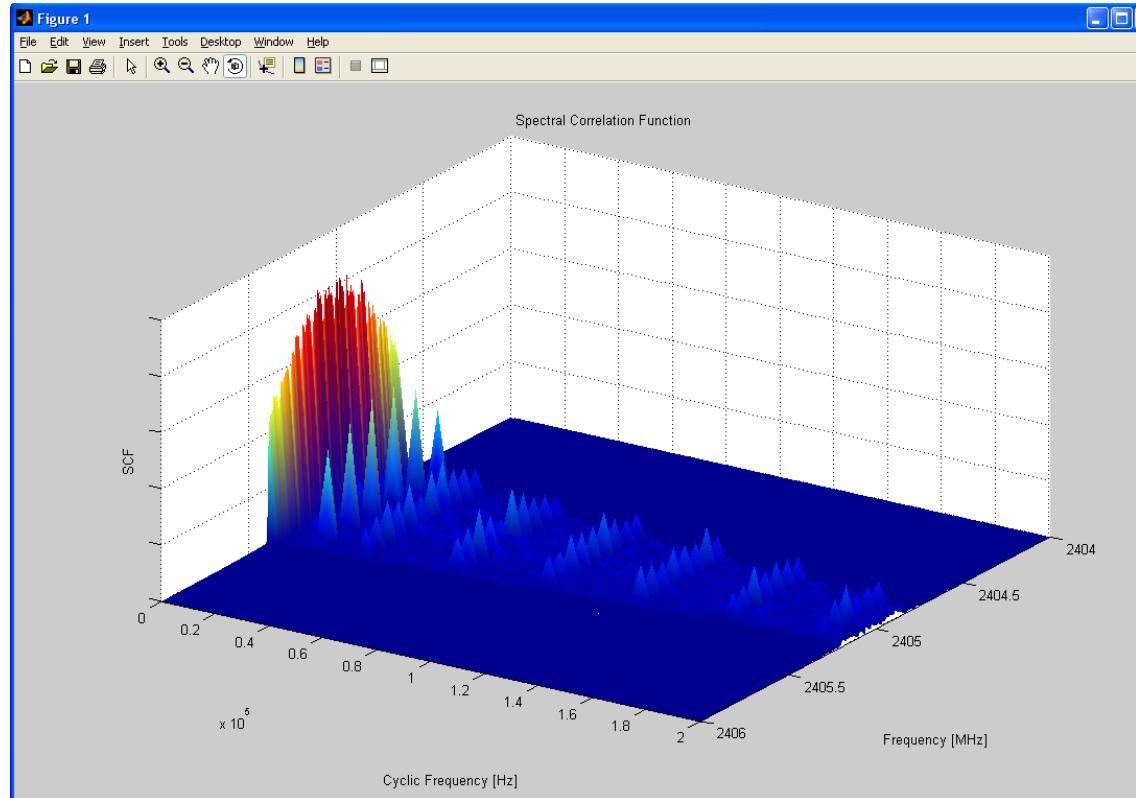
- Network modelling, planning and optimisation
- Basestation positioning, frequency planning, transmit power control



<http://www.awe-communications.com/>

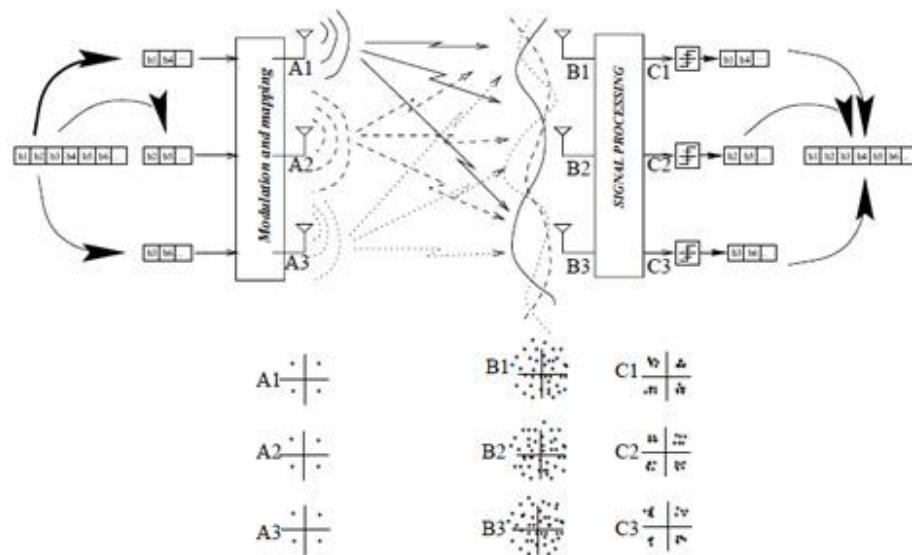
# Applications in Wireless Comm.

- 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

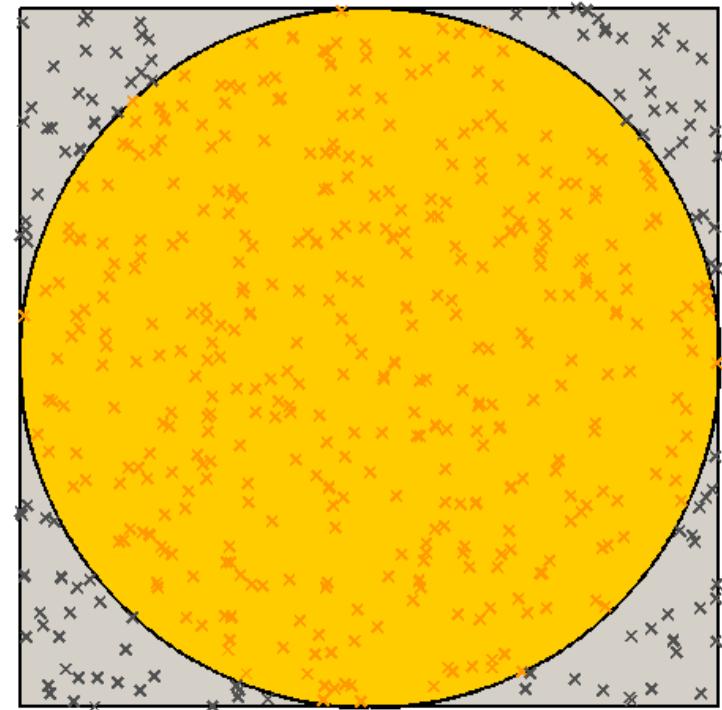
- De-facto standard API for writing shared memory parallel applications in C, C++, and Fortran for CPUs
- Consists of:
  - Compiler directives
  - Run time routines
  - Environment variables
- Specification maintained by the OpenMP Architecture Review Board
- Need to explicitly address parallelism → 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

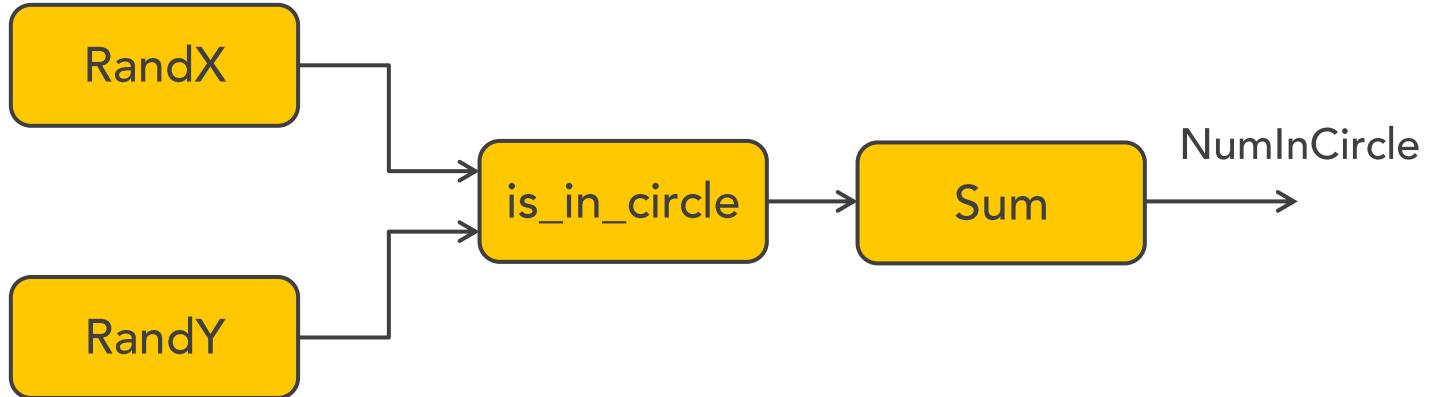
- Monte-Carlo algorithm for estimating the value of Pi
- Draw a square on the ground
- Draw a circle within it
- Scatter grains of rice over the square
  - Count the number of grains within the circle
  - $\pi/4 = \text{grains in circle} / \text{total grains}$



$$\pi = 4 \frac{N_{\text{in\_circle}}}{N_{\text{total}}}$$

## Example: Monte-Carlo Pi

- Flow graph for the algorithm:



$$\pi = 4 * \text{NumInCircle} / \text{Total}$$

# OpenMP Example: Monte-Carlo Pi

```
... // 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 ;
}
```

directives

API calls

# CUDA Overview

- Nvidia standard for GPGPU programming
  - Uses a dialect of C with extensions and restrictions
  - Need to explicitly handle CPU  $\Leftrightarrow$  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

# CUDA Example: Monte-Carlo Pi

```
... //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++;
        else {}
    }
    __syncthreads();
}
```

1/2

```
if (z <= 1) incircle++;
else {}
}
__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, sizeof(float),
cudaMemcpyDeviceToHost);

    *solution = 4*(sumHost[0]/total);
    printf("%.4f\n", 1000, *solution);
    free (solution);

    return 0;
}
```

2/2

# Programming Challenge

- Great hardware potential

Intel® Core™ i7-3960X	Nvidia Tesla M2090
<ul style="list-style-type: none"><li>• 6 Cores</li><li>• 12 threads</li><li>• 70 GFLOPS (DP)</li></ul> <p>(Source: Linpack benchmark)</p>	<ul style="list-style-type: none"><li>• 512 CUDA Cores</li><li>• Hundred1000s of threads</li><li>• 665 GFLOPS (DP)</li></ul> <p>(source: <a href="http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf">http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf</a>)</p>

- 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

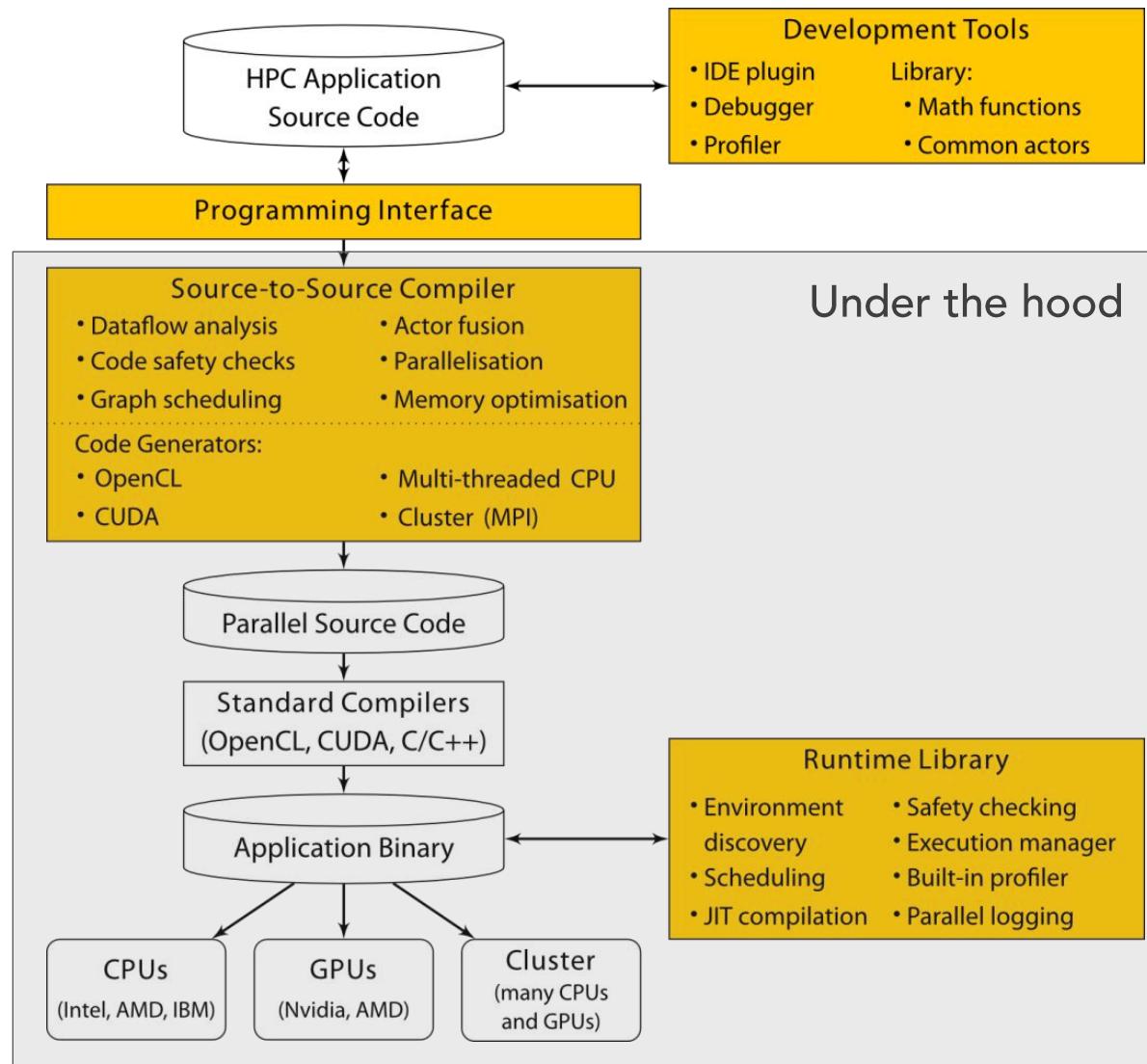
# Generic Software Development Tools

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

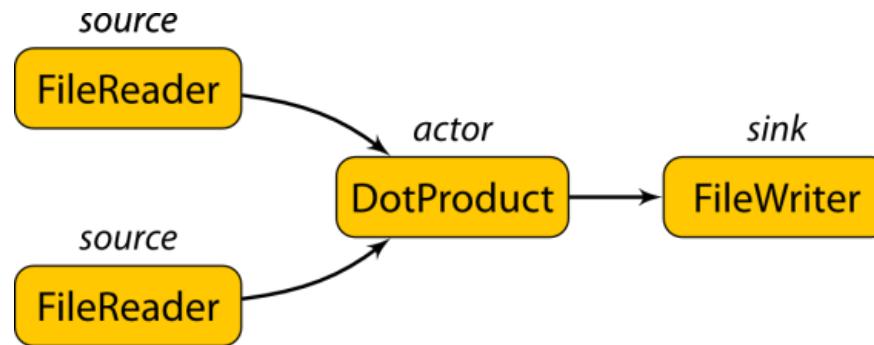
# Xcelerit SDK Overview

- <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



# Xcelerit SDK Programming Model



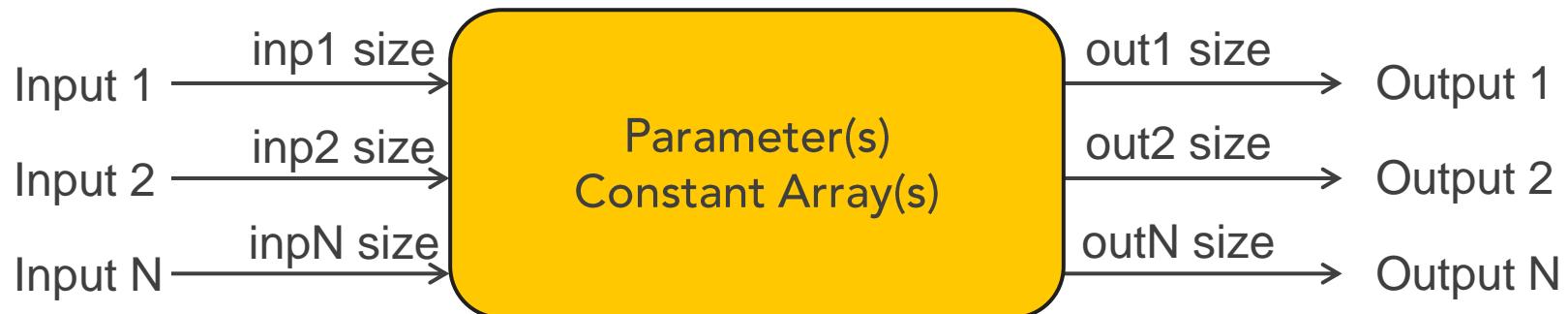
*actor* has inputs and outputs, processes streams of inputs

*source* provides data streams for graph

*sink* consumes final outputs

# Xcelerit SDK Programming Interface: Actors

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



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

# Actor Example: Adder

```
// declare actor interface
actor Adder {
    __input float a[1];
    __input float b[1];
    __output float sum[1];
};

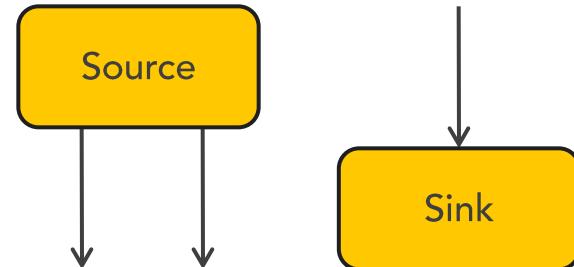
// core algorithm
actor_run<Adder>() {
    sum[0] = a[0] + b[0];
}
```

- 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



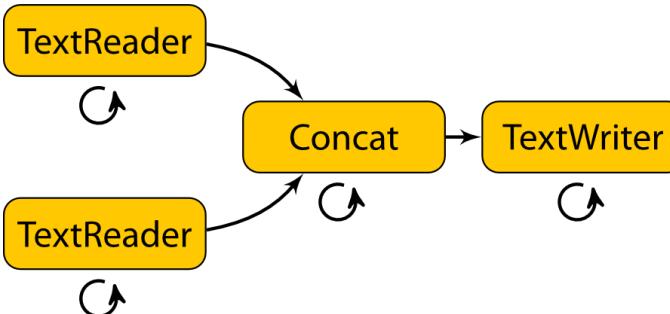
```
class MySource : public Source { ...  
class MySink : public Sink { ...
```

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

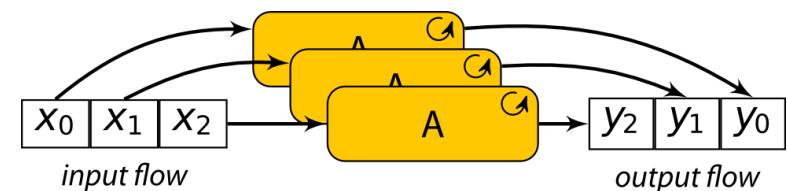
```
out1.setProductionRate(123);  
in.setConsumptionRate(42);
```

```
void run()  
{ /* produce/consume data */ }
```

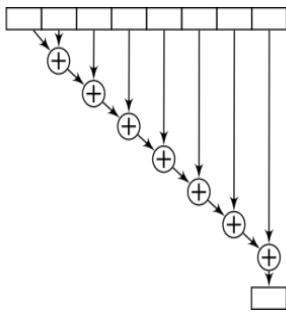
# Under the Hood: Automatic Parallelism Extraction



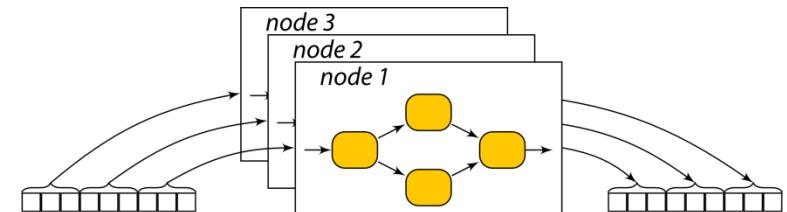
Pipeline Parallelism



Data parallelism (incl. SIMD)



Parallel Reductions



Cluster: data parallelism

# Xcelerit SDK: Monte-Carlo Pi

## Sequential C++ (Traditional)

```
// 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;
}

... // 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

```
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;
}
```

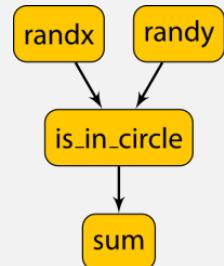
```
... // includes
#define N 1000000

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 );

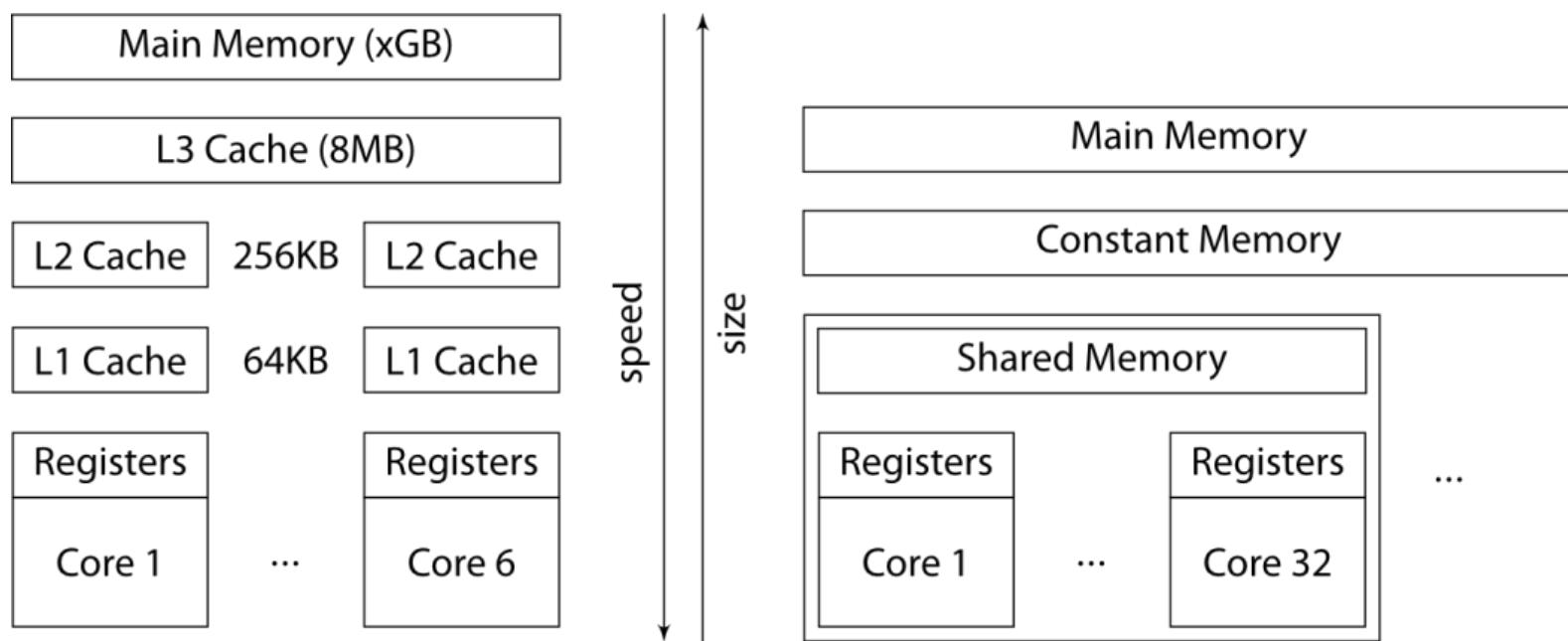
    // connect dataflow graph
    Flowgraph f;
    f += randx >> is_in_circle.x ,
        randy >> is_in_circle.y ,
        is_in_circle.out >> sum ;
    f.run();

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



# Xcelerit SDK Under the Hood: Automatic Optimisation and Error Checks

- 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

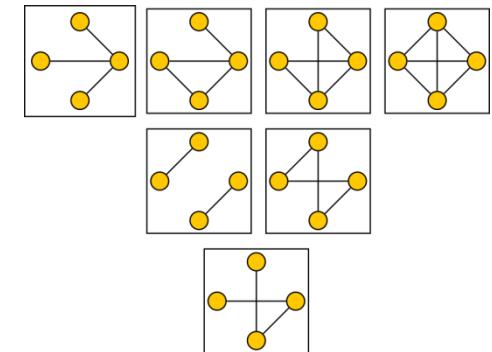
- Intuitive way of thinking for mathematicians, scientists in many domains
- Application-centric - No parallel constructs
- Completely abstract the hardware - No need to know what runs on what
- Other hardware like FPGAs, DSPs, can be supported in future with same source code
- Automatic optimisation: 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  $N = 3, 4, \dots, 7$  nodes
  - Monte-Carlo with 100,000 paths for each topology
  - Each path: Game Theory simulation, testing if learning algorithm converges



All topologies with 4 nodes.

Nodes	3	4	5	6	7
Topologies	2	7	23	122	888

# Xcelerit SDK Network Simulation

## Sequential C++ (Traditional)

```
... // 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] = ...;
}
```

```
... // 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

```
... // 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] = ...;
}
```

```
... // defines and includes

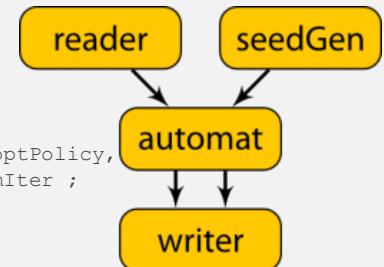
class GraphReader : public Source { ... };
class Writer : public Sink { ... };
randseed randSeed () { ... }
```

```
int main () {
    xceleritSdk sdk; // initialise SDK

    // instantiate sources , sinks and actors
    GraphReader reader("graphs.dat");
    SrcFunctor<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 ;

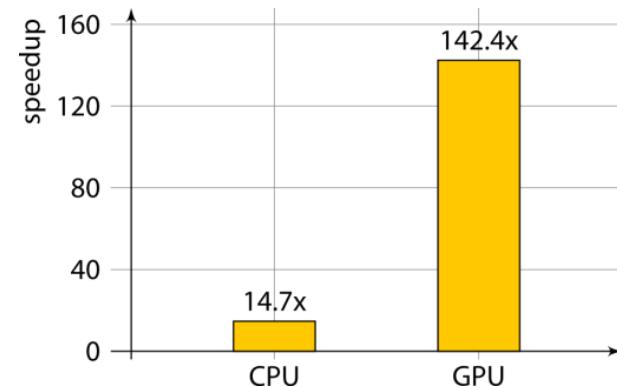
    f.run (); // run flow graph
}
```



# 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)

Nodes	Topolo-gies	Sequen-tial	Xcelerit		Speedup	
			CPU	GPU	CPU	GPU
3	2	00:01:40	00:00:06	00:00:01	15.9	141.8
4	7	00:08:37	00:00:37	00:00:04	14.1	147.2
5	23	00:36:27	00:02:41	00:00:16	13.6	133.5
6	122	03:56:48	00:17:49	00:01:45	13.3	135.0
7	888	34:32:00	02:20:56	00:14:33	14.7	142.4



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



# Thank You

Email: lahlou (dot) hicham (at) gmail (dot) com  
LinkedIn: <http://www.linkedin.com/in/hichamlahlou>