Evolutionary Computation With GPUs

Pierre Collet

SONIC
Stochastic Optimisation and
Nature Inspired Computing

BFO – LSIIT
Université De Strasbourg
pierre.collet@unistra.fr

Simon Harding

IDSIA
SUPSI, USI
Lugano, Switzerland

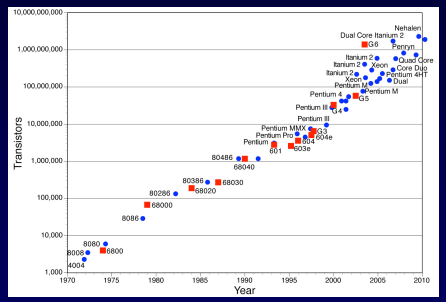
Organisation of the tutorial

- General scope (P. Collet)
  - GPGPU Super-computers
  - Clusters of GPGPU machines
- GPU Programming 101 (S. Harding)
  - Programming model
  - CUDA 101
  - Alternatives to Cuda
- EASEA platform (P. Collet)
  - Benchmarks and real problems

AMD Phenom II X4 940 BE running at 6Ghz
in a bath of liquid nitrogen at -195°C...

Moore's Law

- Doubling of transistor density every 2 years.



- Will hold until 2029 according to Pat Gelsinger (Intel)

GPUs are coming fast

- One slide presented at SC'10 by Bland (Oak Ridge):

Results of Application Readiness Review of Titan Accelerator-Based system

- "Use of the GPU did lead to a performance relative to power cost improvement in almost all cases."
- "There is significant upside potential in GPU performance as we learn how to effectively use manycore architectures and develop new algorithms."
- "GPUs are a harbinger of all future processors to come and there is ample evidence that designing applications for today's GPUs will positively impact the performance of all multicore and manycore processors both today and in the future."
- "Giving OLCF users access to a machine that is competitive as both a CPU and GPU system will provide an excellent transition vehicle for manycore applications development."

http://computing.ornl.gov/SC10/documents/SC10_Booth_Talk_Bland.pdf

Future computers will be massively parallel

- In Nov 2007, the fastest machine was 500 Tflops (212,992 PowerPC 440 cores clocked at 700MHz).
- In Oct 2010, Tianhe-1a is capable of 2.5 Pflops. It is composed of 112 computer cabinets, 12 storage cabinets, 6 communications cabinets, and 8 I/O cabinets, for a total of 3,584 blades, containing 14,336 2.93GHz CPUs and 3 million 575MHz GPU cores.
- March 2011: Oak Ridge National Lab launches TITAN, a 20 Pflop machine based on NVIDIA GPU cards.
- Exaflop machines are predicted to appear by 2020, and Zetaflop machines by 2030.

Question : how do you parallelize programs to efficiently use such machines ?

EC and massive parallelism

- Evolutionary Algorithms are *generic* optimization methods that are intrinsically parallel.
- They can exploit ***ANY*** kind of parallelism (SIMD, SPMD, MIMD, …)
- EA parallelism scales well (possibly supralinear speedup !)
- Perfect paradigm for Peta and Exascale computing, provided one can handle 4 levels of parallelism:
  - Massive parallelism on one GPU card
  - Parallelizing over several GPU cards
  - Parallelizing over several GPU machines
  - Parallelizing over several clusters of GPU machines

GPU PROGRAMMING 101

Getting started

- We need to learn a bit about the hardware
- Which in turn teaches us something about the software engineering approaches
- Which in turn leads on to programming GPUs

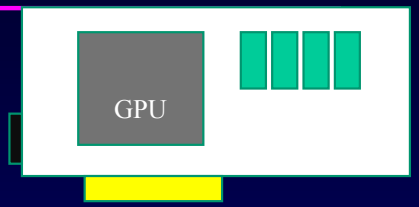
Pierre Collet Simon Harding

Coming up

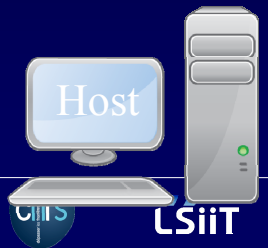
- We will primarily look at CUDA
- But will also mention some alternatives:
  - Shaders
  - Products from GASS, Tidepowrd and Microsoft
  - OpenCL

Pierre Collet Simon Harding

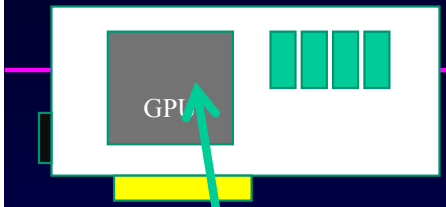
Device

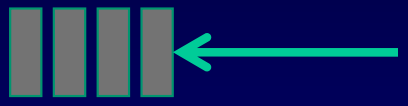


PCI-E bus interface

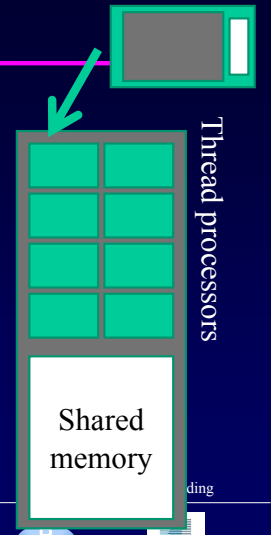


Pierre Collet Simon Harding





Multiprocessor



Pierre Collet

Nvidia 480:

- 15 multiprocessors
- 32 cores per multiprocessor
- 480 cores in total
- 64Kb Shared memory per multiprocessor
- 1.5Gb Global memory

Can you add throughput (bandwidth) ?

What is CUDA?

- C/C++ based language for GPGPU programming
- From Nvidia
- Only works on their graphics cards
- But it's really common in the field...

Programming model

- It's all about threads
  - Thousands of threads are OK
  - Thread is also called a kernel
- Each core runs a thread
- Threads are bundled together in blocks
- Blocks are bundled together in grids

Programming model

- Each thread can see the local memory of that processor core
- Threads within a block see the same shared memory
- Between blocks, there is the global memory

Programming model

- Each thread can calculate an ID using predefined variables available in the kernel:
  - gridDim : Dimensions of the grid in blocks
  - blockDim : Dimensions of the block in threads
  - blockIdx : Block index within the grid
  - threadIdx : Thread index within the block
- Dimensions are defined when the kernels are launched.

CUDA 101

- Let's write a program to add two vectors of numbers together
- E.g. [0,1,2,3,4] + [5,6,7,8,9] = [5,7,9,11,13]
- Do this in parallel
  - Each thread will deal with one index in the vectors

CUDA 101

CUDA threads don't return this way

```
__global__ void add( int *a, int *b, int *c )
```

Means it runs on the device, and can be called from the host

We'll pass pointers to 2 arrays to add together – and a pointer of where to store the results.

CUDA 101

```
__global__ void add( int *a, int *b, int *c )
    {
    int index =
        threadIdx.x + blockIdx.x * blockDim.x;

    c[index] = a[index] + b[index];
    }
```


CUDA 101

- Now we need to get this running...
- Let's write a main method

CUDA 101

```
int main( void )
{
    return 0;
}
```


CUDA 101

```
int main( void )
{
  int DataLength = 2048 * 2048;
     int DataSize = DataLength * sizeof(int) ;
  int *a, *b, *c; //Pointers for HOST SIDE DATA

  a = (int*)malloc(DataSize);
  b = (int*)malloc(DataSize);
  c = (int*)malloc(DataSize);

  //Put some numbers in a and b
```


CUDA 101

```
// Pointers to the device side data
int *dev_a, *dev_b, *dev_c;

cudaMalloc( (void**)&dev_a, DataSize );
cudaMalloc( (void**)&dev_b, DataSize );
cudaMalloc( (void**)&dev_c, DataSize );

cudaMemcpy(dev_a, a, size, cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, b, size, cudaMemcpyHostToDevice);
```


CUDA 101

```
int ThreadsPerBlock = 512;
int NumberOfBlocks = DataLength / ThreadsPerBlock;

add<<<NumberOfBlocks, ThreadsPerBlock>>>
    (
    dev_a, dev_b,
  dev_c
  );
```


CUDA 101

```
//Get the results back from GPU memory
cudaMemcpy(
    c,
    dev_c,
    DataSize,
    cudaMemcpyDeviceToHost
    );
```


CUDA 101

```
    //Tidy up
    free( a );
    free( b );
    free( c );
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );

    //Exit!
    return 0;
    }
```


Another CUDA example

- Dot product
  - Takes two vectors, returns one value (reduction!)
- Will demonstrate how to synchronise threads, use shared memory and how to avoid a nasty race condition

Another CUDA example

```
//Define the data size
#define N  (2048*2048)

//Define the block size
#define THREADS_PER_BLOCK 512
```

Pierre Collet Simon Harding

Another CUDA example

```
__global__ void dot( int *a, int *b, int *c )
{
  __shared__ int temp[THREADS_PER_BLOCK];
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  temp[threadIdx.x] = a[index] * b[index];

  __syncthreads();

  if( 0 == threadIdx.x ) {
    int sum = 0;
    for( int i = 0; i < THREADS_PER_BLOCK; i++ )
      sum += temp[i];
    atomicAdd( c , sum );
  }
}
```

Pierre Collet Simon Harding

Another CUDA example

```
__global__ void dot( int *a, int *b, int *c )
{
 __shared__ int temp[THREADS_PER_BLOCK];
 int index = threadIdx.x + blockIdx.x * blockDim.x;
  temp[threadIdx.x] = a[index] * b[index];

  __syncthreads();

  if( 0 == threadIdx.x ) {
    int sum = 0;
    for( int i = 0; i < THREADS_PER_BLOCK; i++ )
      sum += temp[i];
    atomicAdd( c , sum );
  }
}
```

Pierre Collet Simon Harding

Another CUDA example

```
__global__ void dot( int *a, int *b, int *c )
{
  __shared__ int temp[THREADS_PER_BLOCK];
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  temp[threadIdx.x] = a[index] * b[index];

  __syncthreads();

  if( 0 == threadIdx.x ) {
    int sum = 0;
    for( int i = 0; i < THREADS_PER_BLOCK; i++ )
      sum += temp[i];
    atomicAdd( c , sum );
  }
}
```

Pierre Collet Simon Harding

Another CUDA example

```
__global__ void dot( int *a, int *b, int *c )
{
  __shared__ int temp[THREADS_PER_BLOCK];
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  temp[threadIdx.x] = a[index] * b[index];

  __syncthreads();

  if( 0 == threadIdx.x ) {
    int sum = 0;
    for( int i = 0; i < THREADS_PER_BLOCK; i++ )
      sum += temp[i];
    atomicAdd( c , sum );
  }
}
```


Another CUDA example

```
__global__ void dot( int *a, int *b, int *c )
{
  __shared__ int temp[THREADS_PER_BLOCK];
  int index = threadIdx.x + blockIdx.x * blockDim.x;
  temp[threadIdx.x] = a[index] * b[index];

  __syncthreads();

  if( 0 == threadIdx.x ) {
    int sum = 0;
    for( int i = 0; i < THREADS_PER_BLOCK; i++ )
      sum += temp[i];
    atomicAdd( c , sum );
  }
}
```


Why atomicAdd?

- Why not c+=sum?
- c+=sum is not safe – race conditions
  - Read value at c
  - Read value of sum
  - Write new value back to c
- With many things happening at once, the values could be corrupted.
- atomicAdd solves this.

CUDA

- This tutorial only provides the real basics.
- Lots more to learn
  - Especially if you want highly optimized CUDA
- Lots of resources on line:
  - Google is your friend here.

Using CUDA from elsewhere

- If you have existing code you want to leverage
  - Especially in Java, .net, Python
- Or you just hate programming in C
  - Memory management!
- Wrappers maybe the way forward!

According to wikipedia

- There are MANY options:

Language bindings

- Fortran - FORTRAN CUDA, PGI CUDA Fortran Compiler
- Lua - KappaCUDA
- IDL - GPULib
- Mathematica - CUDALink
- MATLAB - Parallel Computing Toolbox and Distributed Computing Server,[16] as well as 3rd party packages like Jacket.
- .NET - CUDA.NET
- Perl - KappaCUDA
- Python - PyCUDA KappaCUDA
- Ruby - KappaCUDA
- Java - jCUDA, JCuda, JCublas, JCufft
- Haskell - Data.Array.Accelerate
- .NET - CUDAfy.NET .NET kernel and host code, CURAND, CUBLAS, CUFFT.

Quick note!

- Nvidia also supply BLAS and FFT libraries
- These are also wrapped in other languages
  - And in fact can be very easy to use from other languages.

```
import numpy from pycublas import CUBLASMatrix
A = CUBLASMatrix( numpy.mat([[1,2,3],[4,5,6]],numpy.float32) )
B = CUBLASMatrix( numpy.mat([[2,3],[4,5],[6,7]],numpy.float32)
C = A*B print C.np_mat()
```

(code from Wikipedia)

pyCuda

```
mod = comp.SourceModule("""
    __global__ void multiply_them(float *dest, float *a, float
*b) { const int i = threadIdx.x; dest[i] = a[i] * b[i]; }
    """)

multiply_them = mod.get_function("multiply_them")
a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)
dest = numpy.zeros_like(a)
multiply_them(drv.Out(dest), drv.In(a), drv.In(b),
    block=(400,1,1))
print dest-a*b
```

pyCuda

- http://mathema.tician.de/software/pycuda

GASS CUDA.NET

http://www.hoopoe-cloud.com/Solutions/CUDA.NET/

- For .net / mono
- Wraps up calls to .cubin files
  - i.e. Can load and run functions from precompiled CUDA kernels
- Has been used for GP

jCuda

http://www.jcuda.de/

- Java wrapper
- Again wraps up calling of CUDA functions
- Lots of examples, well supported

Don't like the look of CUDA?

- What else is there that has been used in the EA community and elsewhere?

Cg

- C for graphics
- http://developer.nvidia.com/page/cg_main.html
- Pretty much obsolete technology
- See paper "A Data Parallel Approach to Genetic Programming using Programmable Graphics Hardware" by Darren Chitty

Cg – D. Chitty

1. Initialise OpenGL and setup graphics window
2. Copy data inputs into separate textures
3. Create a cgContext object
4. Generate the initial population
5. For each individual:
   - Convert chromosome to Cg source code using recursive functions
   - Compile and load program using Cg Toolkit
   - Bind program and textures to the cgContext object
   - Render the Cg program on the data
   - Get output from the GPU
   - Compare result with desired output and assign a fitness to the individual
6. For each generation:
   - Generate a new population using crossover and mutation
   - Evaluate population by performing step 5
7. Cleanup and output best result

Cg – D. Chitty

```
float4 FragmentProgram (in float2 coords : TEXCOORD0,
     uniform samplerRECT InputTextureX,
     uniform samplerRECT InputTextureY,
     uniform samplerRECT InputTextureZ)   : COLOR
{
     float4 X = texRECT (InputTextureX, coords);
     float4 Y = texRECT (InputTextureY, coords);
     float4 Z = texRECT (InputTextureZ, coords);

     float4 V1 = X;
     float4 V2 = X;
     float4 V3 = V1*V2;
     float4 V4 = X;
     float4 V5 = V3*V4;
     float4 V6 = Y;
     float4 V7 = V5-V6;
     float4 V8 = Z;
     float4 V9 = Z;
     float4 V10 = V8*V9;
     float4 V11 = V10+V7;

     return(V11);
}
```


RapidMind

- Obslete
- Company bought up by Intel
- Seems to have moved to new product : Array Building Blocks
- http://software.intel.com/en-us/articles/intel-array-building-blocks/
- See "A SIMD Interpreter For Genetic Programming on Graphics Cards" by W. Langdon and W. Banzhaf

Shader programming

- As the name suggests, this is really about graphics
- But it is possible to abuse for computations
- See “Linear Genetic Programming GPGPU on Microsoft’s Xbox 360” by G. Wilson and W. Banzhaf for an example using HLSL

Pierre Collet Simon Harding

MS Accelerator

- Research platform
- http://research.microsoft.com/en-us/projects/accelerator/
- .net library for vector maths
- See “Fast Genetic Programming & Developmental Systems” by S. Harding and W.Banzhaf

Pierre Collet Simon Harding

MS Accelerator

```
ParallelArrays.InitGPU();

float[,] CPU_Array1 = new float[4096,4096];
float[,] CPU_Array2 = new float[4096, 4096];

//Populate CPU arrays here

FloatParallelArray GPU_Array1 = new DisposableFloatParallelArray(CPU_Array1);
FloatParallelArray GPU_Array2 = new DisposableFloatParallelArray(CPU_Array2);

FloatParallelArray GPU_Array3 = ParallelArrays.Add(GPU_Array2, GPU_Array2);

FloatParallelArray GPU_Array4 = ParallelArrays.Divide(0.1234f, GPU_Array3);

float[,] CPU_Result = new float[4096, 4096];

ParallelArrays.ToArray(GPU_Array4, out CPU_Result);

// Process CPU_Result array here

ParallelArrays.UnInit();
```

GPU.NET

- Commercial product from TidePowerd
- http://www.tidepowerd.com/
- Converts compiled .net code to device code
- See new paper tomorrow in the CIGPU session : “Implementing Cartesian Genetic Programming Classifiers on Graphics Processing Units using GPU.NET” by S. Harding and W. Banzhaf

Pierre Collet Simon Harding

GPU.Net

```
[ Kernel ]
private static void AddGpu(float[] a, float[] b, float[] c)
{
  // Get the thread id and total number of threads
  int ThreadId = BlockDimension.X * BlockIndex.X +
                 ThreadIndex.X;
  int TotalThreads = BlockDimension.X * GridDimension.X;

  // Loop over the vectors 'a' and 'b', adding them
  // pairwise and storing the sums in 'c'
  for ( int ElementIndex = ThreadId; ElementIndex <
            a.Length; ElementIndex += TotalThreads)
  {
    c[ElementIndex] = a[ElementIndex] + b[ElementIndex];
  }
}
```


GPU.Net

```
// 1. Create a standard .NET
//    array of integers
const int Count = 0x1000000
float[] a =new float[Count];
float[] b =new float[Count];

// Create an array to hold
// the output values
float[] c =new float[Count];

// 2. Set grid/block size
//    for GPU execution
Launcher.SetGridSize(256);
Launcher.SetBlockSize(128);

// 3. Call the kernel method
AddGpu(a, b, c);
```


OpenCL

- http://www.khronos.org/opencl/
- Open standard for GPU and parallel programming
  - Multivendor
  - Diverse hardware (CPU, FPGA, GPU etc)
- See competition entry “CUDA and OpenCL-based asynchronous PSO” by Y. Nashed et al.

OpenCL

- Still relatively unexplored by the community
- Wrappers exist for access from other languages
- Other tech uses it. See WebCL

OpenCL

```
__kernel void vector_add_gpu (__global const float* src_a,
                              __global const float* src_b,
                              __global float* res,
                              const int num)
{
   /* get_global_id(0) returns the ID of the thread in execution.
   As many threads are launched at the same time, executing the same kernel,
   each one will receive a different ID, and consequently perform a different computation.*/
   const int idx = get_global_id(0);

   /* Now each work-item asks itself: "is my ID inside the vector's range?"
   If the answer is YES, the work-item performs the corresponding computation*/
   if (idx < num)
      res[idx] = src_a[idx] + src_b[idx];
}
```

Pierre Collet Simon Harding

http://opencl.codeplex.com/wikipage?title=OpenCL%20T

What does the future hold?

- Who knows?
- Hardware: Intel multiple cores, AMD's APUs
- Software: TPL, New updates to CUDA, OpenCL

Pierre Collet Simon Harding

Different kinds and levels of parallelism

- Low-level massive parallelism:
  - GeForce GTX 580 contains 512 cores running in an SPMD model :
    - One single program (one context).
    - Cores grouped by 32, running in SIMD mode, but different bundles can run on different parts of the code.
  - Several cards in one machine : MIMD model.
- High-level parallelism:
  - Several machines together (cluster of machines).
  - Several clusters together.

Pierre Collet Simon Harding 59

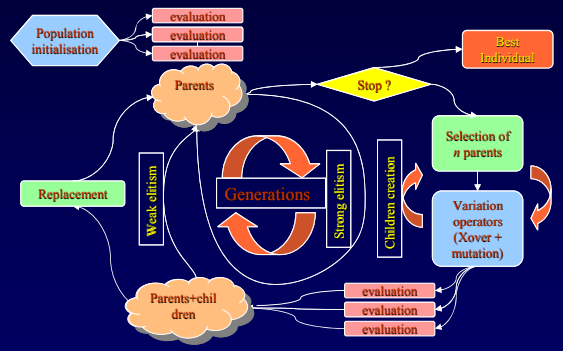
EASEA: an evolutionary platform for massive parallel machines

- First version of EASEA (EAsy Specification of Evolutionary Algorithms) presented ad EA'99
- 2000-2003: EASEA is the programming language of the DREAM (Distributed Resource Evolutionary Algorithm Machine).
- 2008- EASEA is an evolutionary platform to automatically parallelize evolutionary algorithms on 4 levels of parallelism:
  - parallelization on one GPU card,
  - parallelization on several GPU cards,
  - parallelization on several heterogeneous machines
  - parallelization on several heterogeneous clusters of GPU machines.
- 2012 EASEA will run on the French Grid

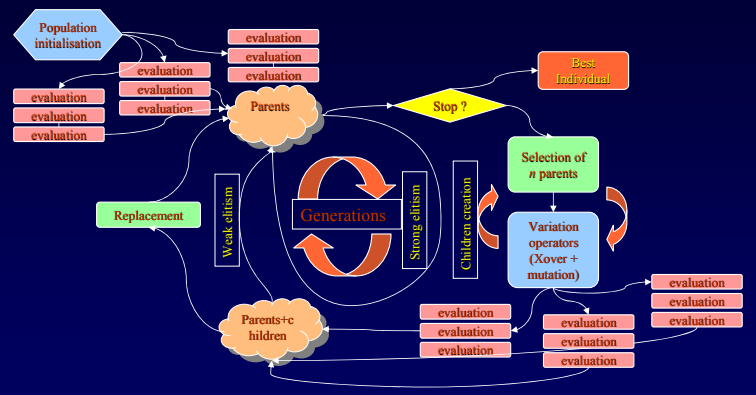
Pierre Collet Simon Harding 60

Low-level massive parallelism (GPU card)

- EASEA generic parallelization over GPU cards.



3 cards in the same PC = x3 speedup !



EASEA : a language that can handle GPGPUs

- EASEA stands for Easy Specification of Evolutionary Algorithm.
- In order to code an evolutionary algorithm in EASEA, the user only needs to write (in C) the application-specific code, i.e. :
  - Initialisation function (how to initialise the genome)
  - Evaluation function (dependent on the problem)
  - Crossover operator (how to recombine 2 genomes)
  - Mutation operator (how to mutate the genome).

*EASEA is available on Sourceforge: http://sourceforge.net/projects/easea/

Weierstrass initialisation and evaluation code

```
\GenomeClass::initialiser :
  for(int i=0; i<N; i++) {
       Genome.x[i] = random((float)-1,(float)1);
  }
```

```
\GenomeClass::evaluator :
  float res = 0.;
  float val[N];
  for (int i = 0;i<N; i++) {
    val[i] = 0.;
   for (int k=0;k<ITER;k++)
      val[i] += pow(2.,-k*.25)
                *sin(pow(2.,k)*Genome.x[i]);
      res += Abs(val[i]);
  }
  return (res);
```

Weierstrass crossover and mutation code

```
\GenomeClass::crossover :
  for (int i=0; i<N; i++) {
    float alpha = random(0.,1.);
    child->Genome.x[i] = alpha*parent1->Genome.x[i]
                  +(1.-alpha)*parent2->Genome.x[i];
  }
```

```
\GenomeClass::mutator :
  for (int i=0; i<N; i++)
    if (tossCoin(pMutPerGene)){
      Genome.x[i]+=SIGMA*random((float)0,(float)1);
      Genome.x[i]=MAX(X_MIN,MIN(X_MAX,Genome.x[i]));
    }
```


Evolutionary Algorithm parameters

- Standard EA parameters need to be provided:

```
\Default run parameters :
 Number of generations : 100
  Mutation probability : 1
  Crossover probability : 1
  Population size : 20000
  Genitors selector: Tournament 7
  Offspring size : 100%
  Competing Parents : 50%
  Parents reduce : Deterministic
  Final reduce: Deterministic
  Elitism : On
  Evaluator goal : Minimise
```


EASEA compilation and execution

- Then, typing:

```
$ easea weierstrass -cuda
```

  on the command line will create C++ code for the evolutionary algorithm, and parallelized C code for the CUDA sdk.

- A makefile is automatically generated, so simply typing:

```
$ make
```

  will compile for the GPGPU.

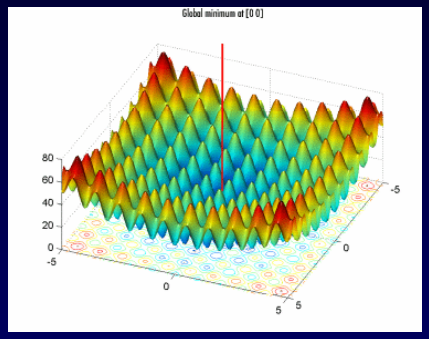
- Then, typing:

```
$ ./weierstrass
```

  will launch the optimisation of the Weierstrass function on the CPU, with parallel evaluations on the GPGPU card

- Thanks to CUDA, all GPGPU NVIDIA cards are supported.

Island model parallelisation

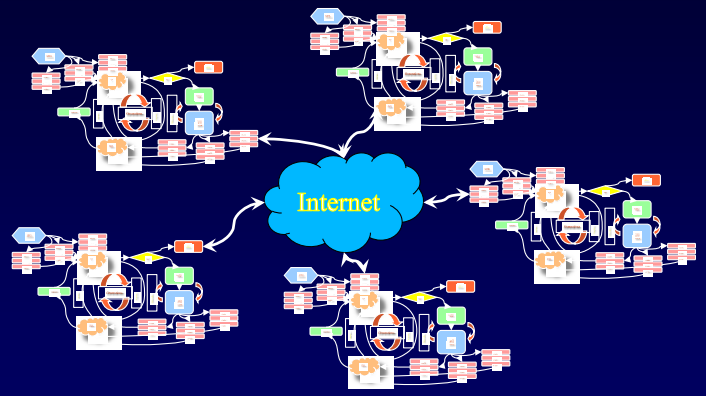


$$Ras(x) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2).$$

Island parallelism

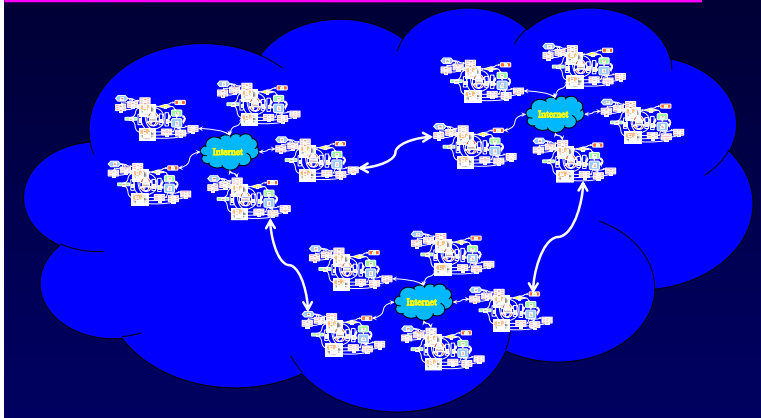
Pierre Collet Simon Harding 69

High-level massive parallelism (island model)



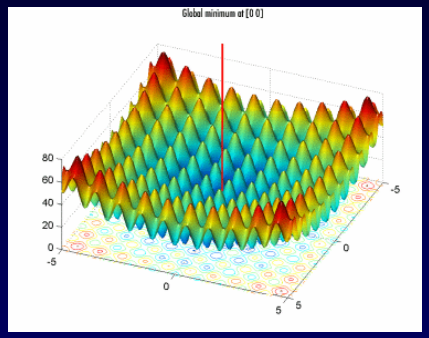
Pierre Collet Simon Harding 70

Many PF machines for Exascale Computing ?



Pierre Collet Simon Harding 71

Benchmark for the island model: Rastrigin

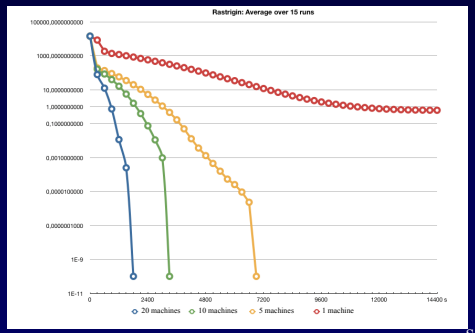


$$Ras(x) = 20 + x_1^2 + x_2^2 - 10(\cos 2\pi x_1 + \cos 2\pi x_2).$$

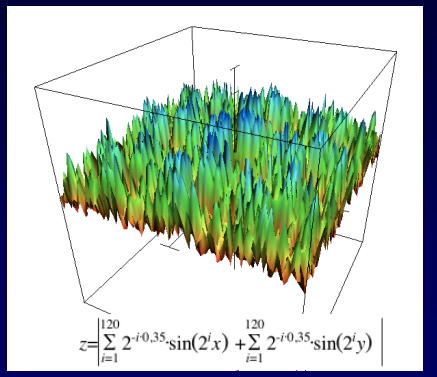
Pierre Collet Simon Harding 72

Island model speedup on Rastrigin-1000

Linear speedup with the number of machines



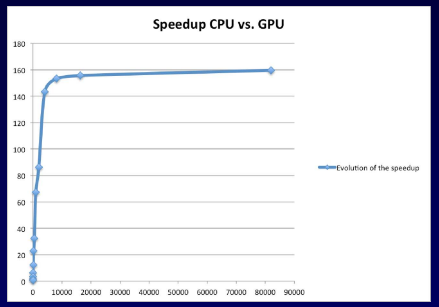
Weierstrass function (h=0.35, 2 dimensions)



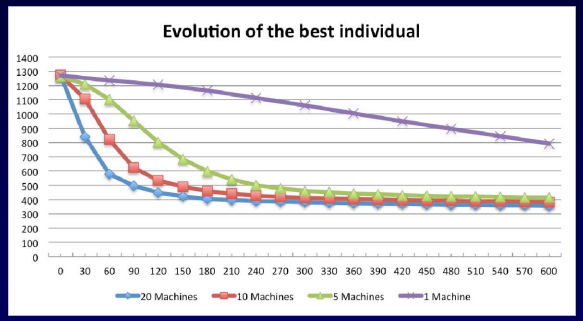
$$z=\left|\sum_{i=1}^{120} 2^{-i\cdot 0,35}\cdot\sin(2^i x)+\sum_{i=1}^{120} 2^{-i\cdot 0,35}\cdot\sin(2^i y)\right|$$

GTX275 *vs* Core i7 950 speedup

- GPU/CPU speedup on 1000 dimensions Weierstrass h=0.35.

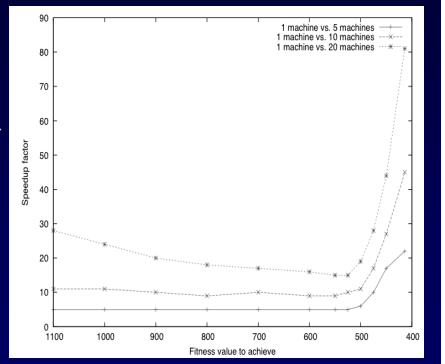


Island model on Weierstrass h=0.35 1000 dim



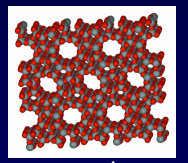
Island model speedup on Weierstrass h=0.35

- Linear speedup until value 525, where one machine only stalls in a local optimum (average over 20 runs).
- Beyond, the island model brings supra-linear speedup, but still in a linear way (on value 425, speedup of 22 for 5 machines, 45 for 10 machines, 82 for 20 machines)
- Impossible to find values under 425 with one machine only while this is reached in 10 mn on 20 machines.
- For value 425, the actual speedup obtained with 20 machines over a single core of core i7-950 is 160x80=12800 ! 1 day = 35 years !

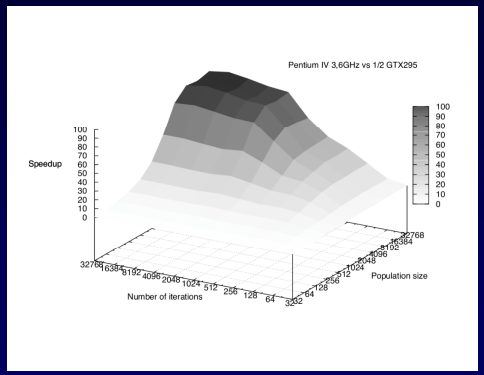


Combining LL and HL massive parallelism

- Zeolite hunt:
  - Zeolites are porous crystalline structures with many applications in industry made of oxygen atoms around a silicon or aluminium atom.
  - A replication of the unit structure allows to obtain the pores.
  - Zeolites are used for filtering, catalysis, energy storing, medicine, absorbing liquids, odours (cat litter), …

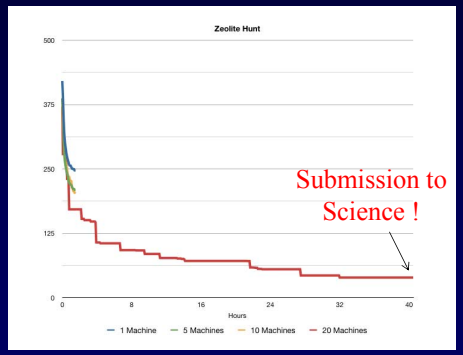


GPU speedup on zeolite problem: ~120



GPU Islands speedup on zeolite problem

- 1, 5, 10 and 20 GPU machines



EC is ready for EC !

- EC is a ***generic*** massively parallel optimization method that can exploit peta and exascale computing.
- Not rocket science anymore: EASEA regularly runs on 20 machines with 256 cores = 5000 cores on many different problems (cf. Zeolite problem)
- Developments: new algorithms must be coined for HUGE populations (100K to 1M individuals on one island).
- New practices must be developed (increasing mutation, dealing with heterogeneous machines, ALPS-like algorithms, …)
- Current problem: finding large enough problems to get such machines to heat up.

More GPU at GECCO

- Tomorrow:
  - CIGPU - Computational Intelligence using Consumer Hardware
- Thursday:
  - Parallel Evolutionary Systems
- Friday:
  - GPU Competition

EA,GPU elsewhere

This work has been sponsored by…

Massively Parallel EC on GPGPUs