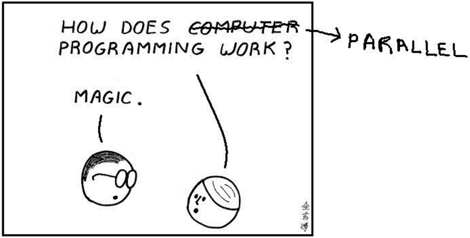


The ‘Bones’ Source-to-Source Compiler: Making Parallel Programming Easy

Cedric Nugteren

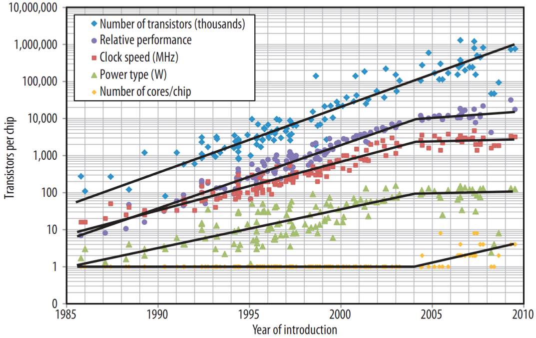
Eindhoven University of Technology (TU/e)
http://parse.ele.tue.nl/
c.nugteren@tue.nl

June 28, 2012

The end of the single-core era

Microprocessor architecture is changing:

- The single-core era has ended...
- ...and makes place for the parallel and heterogeneous computing era



[image taken from '*Computing Performance: Game Over or Next Level?*' by Fuller et al.]

Programming becomes increasingly difficult

The end of the single-core era

The future will see more parallelism

In a few years, everybody will have to program for tens, hundreds or even thousands parallel compute cores

The future will see more heterogeneity

This is how a future processor could look like:

- 100s of lightweight integer units
- 1000s of dedicated floating point units
- A few general purpose cores
- Accelerator hardware for video, network, etc.
- Programmable logic (FPGA soft cores?)

Programming will become even more difficult in the future

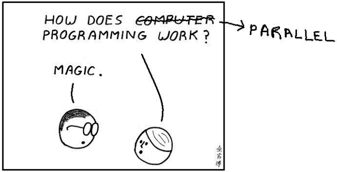
Outline

1. The importance of parallel programming
2. Programming a GPU automatically
3. Introducing ‘algorithmic species’ and ‘Bones’
4. Experimental results
5. Conclusions and future work

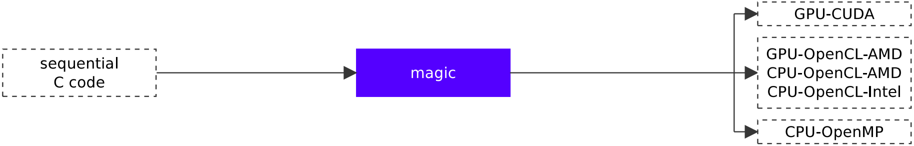
Outline

1. The importance of parallel programming
2. Programming a GPU automatically
3. Introducing ‘algorithmic species’ and ‘Bones’
4. Experimental results
5. Conclusions and future work

Programming a GPU automatically



Our ultimate goal:



sequential C code

magic

GPU-CUDA

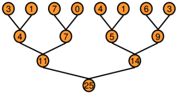
GPU-OpenCL-AMD
CPU-OpenCL-AMD
CPU-OpenCL-Intel

CPU-OpenMP

Example C to CUDA transformation

Example 1: Sum

```
int sum = 0;
for (int i=0;i<N;i++) {
  sum = sum + in[i];
}
```



[image taken from '*Optimizing Parallel Reduction in CUDA*' by Mark Harris]

Just a part of the optimized CUDA code:

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sm, unsigned int tid) {
  if (blockSize >= 64) sm[tid] += sm[tid + 32];
  if (blockSize >= 32) sm[tid] += sm[tid + 16];
  if (blockSize >= 16) sm[tid] += sm[tid +  8];
  if (blockSize >=  8) sm[tid] += sm[tid +  4];
  if (blockSize >=  4) sm[tid] += sm[tid +  2];
  if (blockSize >=  2) sm[tid] += sm[tid +  1];
}

template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
  extern __shared__ int sm[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockSize*2) + tid;
  unsigned int gridSize = blockSize*2*gridDim.x;
  sm[tid] = 0;
  while (i < n) {
    sm[tid] += g_idata[i]
    sm[tid] += g_idata[i+blockSize];
    i += gridSize;
  }
  __syncthreads();
  if (blockSize >= 512) {
    if (tid < 256) { sm[tid] += sm[tid + 256]; }
    __syncthreads();
  }
  if (blockSize >= 256) {
    if (tid < 128) { sm[tid] += sm[tid + 128]; }
    __syncthreads();
  }
  if (blockSize >= 128) {
    if (tid <  64) { sm[tid] += sm[tid +  64]; }
    __syncthreads();
  }
  if (tid < 32) { warpReduce<blockSize>(sm, tid); }
  if (tid == 0) { g_odata[blockIdx.x] = sm[0]; }
}
```

What about a second example?

Example 1: Sum

```
int sum = 0;
for (int i=0;i<N;i++) {
  sum = sum + in[i];
}
```

Example 2: Max

```
int max = 0;
for (int i=0;i<N;i++) {
  max = (max>in[i]) ? max : in[i];
}
```

CUDA code for example 2:

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sm, unsigned int tid) {
  if (blockSize >= 64) sm[tid] = (sm[tid]>sm[tid+32]) ? sm[tid] : sm[tid+32];
  if (blockSize >= 32) sm[tid] = (sm[tid]>sm[tid+16]) ? sm[tid] : sm[tid+16];
  if (blockSize >= 16) sm[tid] = (sm[tid]>sm[tid+ 8]) ? sm[tid] : sm[tid+ 8];
  if (blockSize >=  8) sm[tid] = (sm[tid]>sm[tid+ 4]) ? sm[tid] : sm[tid+ 4];
  if (blockSize >=  4) sm[tid] = (sm[tid]>sm[tid+ 2]) ? sm[tid] : sm[tid+ 2];
  if (blockSize >=  2) sm[tid] = (sm[tid]>sm[tid+ 1]) ? sm[tid] : sm[tid+ 1];
}

template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
  extern __shared__ int sm[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockSize*2) + tid;
  unsigned int gridSize = blockSize*2*gridDim.x;
  sm[tid] = 0;
  while (i < n) {
    sm[tid] = (sm[tid]>g_idata[i]) ? sm[tid] : g_idata[i];
    sm[tid] = (sm[tid]>g_idata[i+blockSize]) ? sm[tid] : g_idata[i+blockSize];
    i += gridSize;
  }
  __syncthreads();
  if (blockSize >= 512) {
    if (tid < 256) { sm[tid] = (sm[tid]>sm[tid+256]) ? sm[tid] : sm[tid+256]; }
    __syncthreads();
  }
  if (blockSize >= 256) {
    if (tid < 128) { sm[tid] = (sm[tid]>sm[tid+128]) ? sm[tid] : sm[tid+128]; }
    __syncthreads();
  }
  if (blockSize >= 128) {
    if (tid <  64) { sm[tid] = (sm[tid]>sm[tid+ 64]) ? sm[tid] : sm[tid+ 64]; }
    __syncthreads();
  }
  if (tid < 32) { warpReduce<blockSize>(sm, tid); }
  if (tid == 0) { g_odata[blockIdx.x] = sm[0]; }
}
```

What about a second example?

Example 1: Sum

```
int sum = 0;
for (int i=0;i<N;i++) {
  sum = sum + in[i];
}
```

Example 2: Max

```
int max = 0;
for (int i=0;i<N;i++) {
  max = (max>in[i]) ? max : in[i];
}
```

CUDA code for example 1:

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sm, unsigned int tid) {
  if (blockSize >= 64) sm[tid] += sm[tid + 32];
  if (blockSize >= 32) sm[tid] += sm[tid + 16];
  if (blockSize >= 16) sm[tid] += sm[tid +  8];
  if (blockSize >=  8) sm[tid] += sm[tid +  4];
  if (blockSize >=  4) sm[tid] += sm[tid +  2];
  if (blockSize >=  2) sm[tid] += sm[tid +  1];
}

template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n) {
  extern __shared__ int sm[];
  unsigned int tid = threadIdx.x;
  unsigned int i = blockIdx.x*(blockSize*2) + tid;
  unsigned int gridSize = blockSize*2*gridDim.x;
  sm[tid] = 0;
  while (i < n) {
    sm[tid] += g_idata[i]
    sm[tid] += g_idata[i+blockSize];
    i += gridSize;
  }
  __syncthreads();
  if (blockSize >= 512) {
    if (tid < 256) { sm[tid] += sm[tid + 256]; }
    __syncthreads();
  }
  if (blockSize >= 256) {
    if (tid < 128) { sm[tid] += sm[tid + 128]; }
    __syncthreads();
  }
  if (blockSize >= 128) {
    if (tid <  64) { sm[tid] += sm[tid +  64]; }
    __syncthreads();
  }
  if (tid < 32) { warpReduce<blockSize>(sm, tid); }
  if (tid == 0) { g_odata[blockIdx.x] = sm[0]; }
}
```

Using algorithmic skeletons for parallel programming

Example 1: Sum

```
int sum = 0;
for (int i=0;i<N;i++) {
  sum = sum + in[i];
}
```

Example 2: Max

```
int max = 0;
for (int i=0;i<N;i++) {
  max = (max>in[i]) ? max : in[i];
}
```

- The examples yield a very similar GPU implementation: They are of the same *class* or '*algorithmic species*'
- Such a GPU implementation is very complex, but only a few lines are different

Idea:

- Make re-use of common code: separate the **structure** from the **functionality**
- The structure is re-used: it is an algorithmic skeleton
- For each class-target combination, there is one piece of skeleton code

Outline

1. The importance of parallel programming
2. Programming a GPU automatically
3. Introducing ‘algorithmic species’ and ‘Bones’
4. Experimental results
5. Conclusions and future work

Introducing Bones (1/2)

A source-to-source compiler with 6 targets:

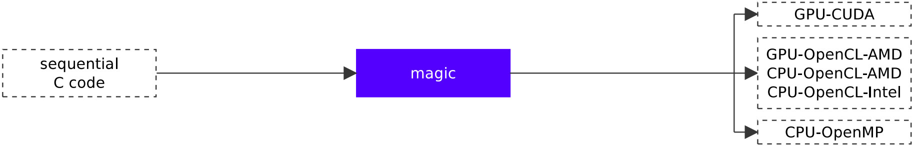
- C-to-CUDA (NVIDIA GPUs)
- C-to-OpenCL (3 targets: AMD GPUs, AMD CPUs, Intel CPUs)
- C-to-OpenMP (multi-core CPUs)
- C-to-C (pass-through)

Bones aims to improve on:

1. Code readability
2. Performance
3. Programmer effort required

Introducing Bones (2/2)

Where does Bones fits into the picture?



sequential C code

magic

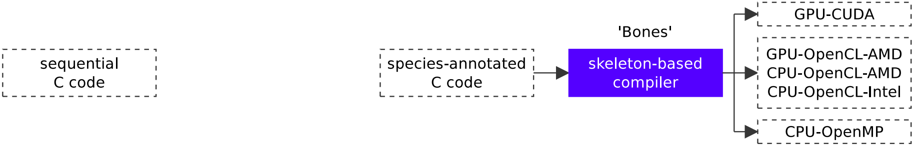
GPU-CUDA

GPU-OpenCL-AMD
CPU-OpenCL-AMD
CPU-OpenCL-Intel

CPU-OpenMP

Introducing Bones (2/2)

Where does Bones fits into the picture?



sequential
C code

species-annotated
C code

'Bones'

skeleton-based
compiler

GPU-CUDA

GPU-OpenCL-AMD
CPU-OpenCL-AMD
CPU-OpenCL-Intel

CPU-OpenMP

Example algorithmic species

0:99,0:15|element → 0:99,0:15|element

```
for (i=0;i<100;i=i+1)
  for (j=0;j<16;j=j+1)
    B[i][j] = 2*A[i][j];
```

0:31|neighbourhood(-1:1) → 0:31|element

```
for (i=0;i<32;i=i+1)
  B[i] = 0.33*(A[i-1] + A[i] + A[i+1]);
```

0:A-1,0:B-1,0:C-1|element → 0:0|shared

```
for (i=0;i<A; i=i+1)
  for  (j=0;j<B; j=j+1)
    for  (k=0;k<C; k=k+1)
      sum[0] = sum[0] + 2*in[i][j][k];
```

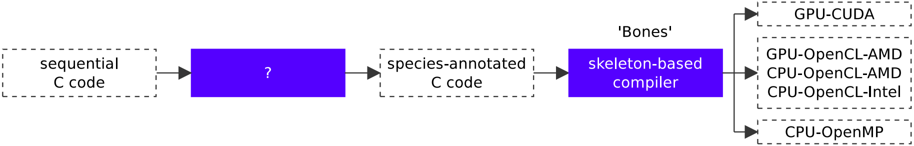
0:7|element ∧ 0:7|element → 0:7|element

```
for (i=0;i<8;i=i+1)
  C[i] = 2*A[i] + B[i];
```

Algorithmic species is an algorithm classification with a formal basis (based on the polyhedral model)

Algorithmic species and Bones

Did we handle all the *magic* yet?

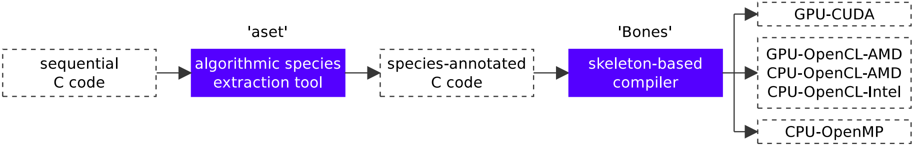


Introducing '*aset*':

- Automatically extract algorithmic species from C-code
- Based on the polyhedral model and algorithmic species theory

Algorithmic species, Bones and aset

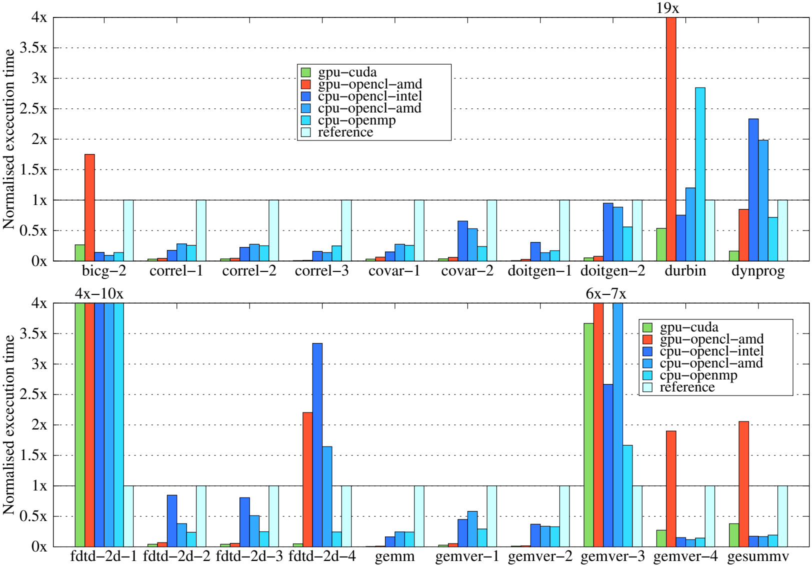
Overview of our complete auto-parallelisation approach:



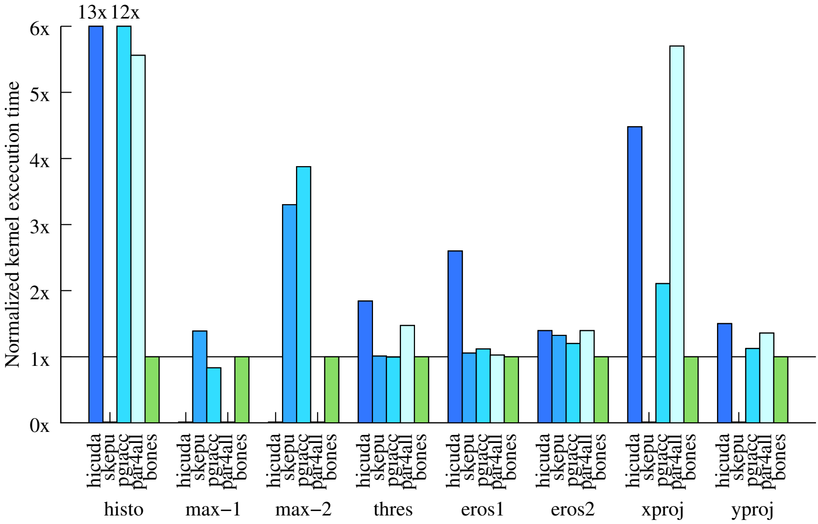
Outline

1. The importance of parallel programming
2. Programming a GPU automatically
3. Introducing ‘algorithmic species’ and ‘Bones’
4. **Experimental results**
5. Conclusions and future work

What do we gain?



How does Bones compare to others?



[Note: this is just a comparison of performance]

Outline

1. The importance of parallel programming
2. Programming a GPU automatically
3. Introducing ‘algorithmic species’ and ‘Bones’
4. Experimental results
5. Conclusions and future work

Conclusions

The new source-to-source compiler Bones:

- Uses the algorithmic skeletons technique
- Generates readable CUDA/OpenCL/OpenMP code
- Delivers competitive GPU performance
- Is based on algorithmic species

The classification 'algorithmic species':

- Captures essential information from C source code
- Is formally defined
- Automates the complete parallelisation process using aset

Future work

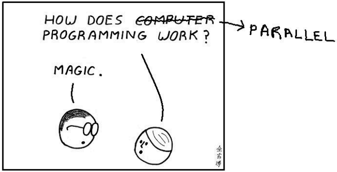
Performance can still be improved:

- Implement and optimise more skeletons
- Perform kernel fusion
- Optimise CPU-GPU data transfers

The work can still be extended further:

- What about irregular algorithms?
- What about multi-GPU and multi-machine code?

Questions / further information



Thank you for your attention!

Bones and aset are available at:
http://parse.ele.tue.nl/bones/
http://parse.ele.tue.nl/species/

For more information and links to publications, visit:
http://parse.ele.tue.nl/
http://www.cedricnugteren.nl/