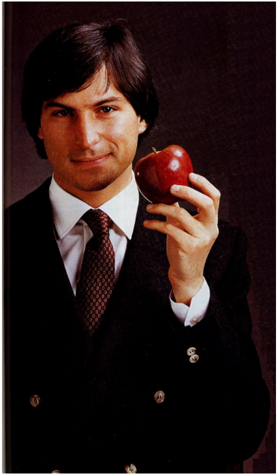
Automatic Skeleton-Based Compilation through Integration with an Algorithm Classification

Cedric Nugteren (presenter), Pieter Custers, Henk Corporaal

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

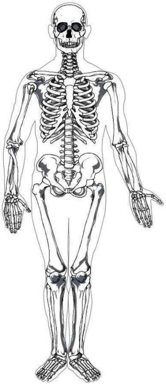
August 28, 2013

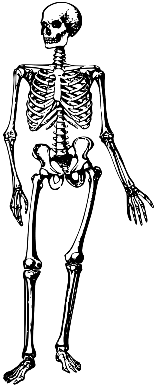
Species and skeletons



Are these two of the same species?

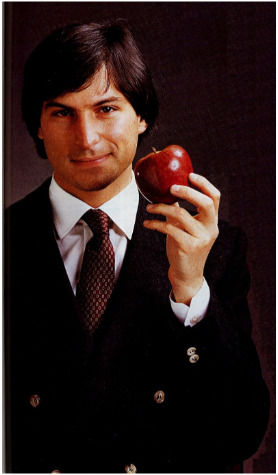
Species and skeletons

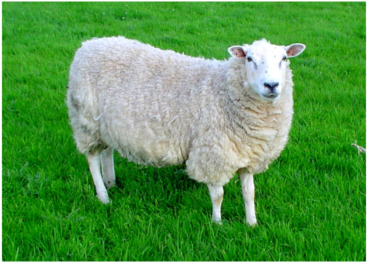




They are. Possible explanation: their skeletons look alike.

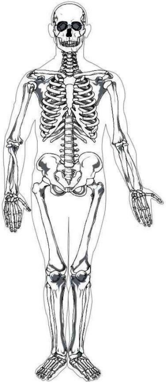
Species and skeletons

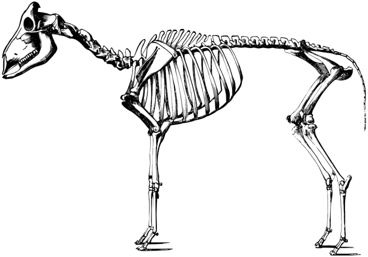




And what about these two?

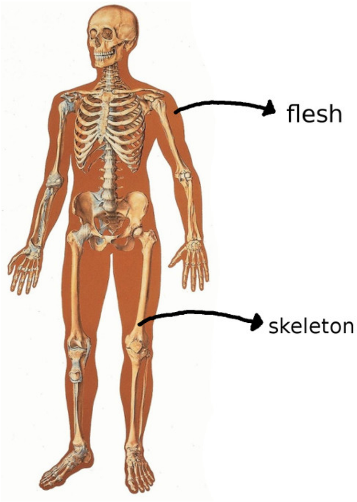
Species and skeletons





They are not: their skeleton is quite different.

Species and skeletons



Functionality of the code: what you want to compute

Structure of the code: parallelism and memory access patterns

Example C to CUDA transformation

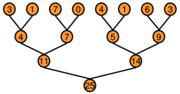
Example 1: Sum

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

Example C to CUDA transformation

Example 1: Sum

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

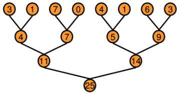


[image taken from '*Optimizing Parallel Reduction in CUDA*' by Mark Harris]

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];
}
```

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];
}
```

- Highlighted is the functionality
- The remainder is the structure: the skeleton of the code

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];
}
```

- Highlighted is the functionality
- The remainder is the structure: the skeleton of the code

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]; }
}
```

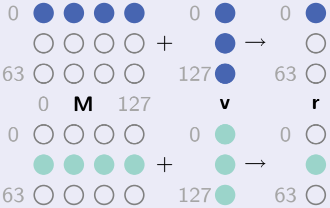
Outline

Outline

Example algorithmic species

Matrix-vector multiplication:

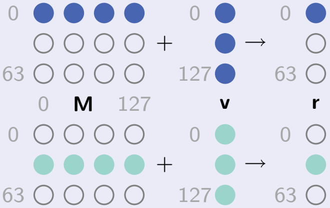
```
for (i=0; i<64; i++) {
  r[i] = 0;
  for (j=0; j<128; j++) {
    r[i] += M[i][j] * v[j];
  }
}
```



Example algorithmic species

Matrix-vector multiplication:

```
for (i=0; i<64; i++) {
  r[i] = 0;
  for (j=0; j<128; j++) {
    r[i] += M[i][j] * v[j];
  }
}
```

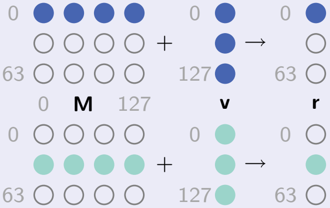


0:63,0:127|chunk(0:0,0:127) ∧ 0:127|full → 0:63|element

Example algorithmic species

Matrix-vector multiplication:

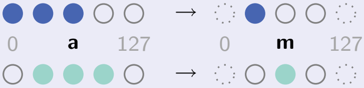
```
for (i=0; i<64; i++) {
  r[i] = 0;
  for (j=0; j<128; j++) {
    r[i] += M[i][j] * v[j];
  }
}
```



0:63,0:127|chunk(0:0,0:127) ∧ 0:127|full → 0:63|element

Stencil computation:

```
for (i=1; i<128-1; i++) {
  m[i] = 0.33 * (a[i-1]+a[i]+a[i+1]);
}
```



1:126|neighbourhood(-1:1) → 1:126|element

Algorithmic species

Algorithmic species:

- Classifies code based on memory access patterns and parallelism
- Is more fine-grained compared to other skeleton classifications
- Can be extracted automatically from C-code using ASET or A-DARWIN

Algorithmic species

Algorithmic species:

- Classifies code based on memory access patterns and parallelism
- Is more fine-grained compared to other skeleton classifications
- Can be extracted automatically from C-code using ASET or A-DARWIN

For more information on species:

1. C. Nugteren, P. Custers, and H. Corporaal. **Algorithmic Species: An Algorithm Classification of Affine Loop Nests for Parallel Programming**. In *ACM TACO: Transactions on Architecture and Code Optimisations, 9(4):Article 40*. 2013.
2. C. Nugteren, R. Corvino, and H. Corporaal. **Algorithmic Species Revisited: A Program Code Classification Based on Array References**. In *MuCoCoS'13: International Workshop on Multi-/Many-core Computing Systems*. IEEE, 2013.

Outline

Introducing Bones (1/2)

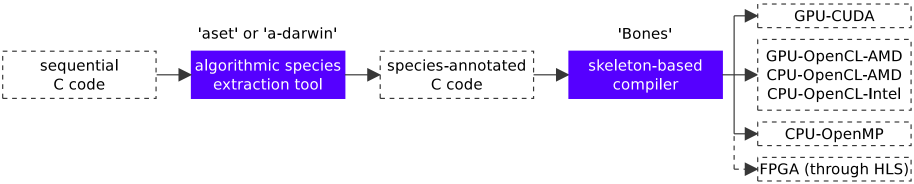
Bones provides a set of skeletons for multiple targets:

- 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)
- C-to-FPGA (under construction)

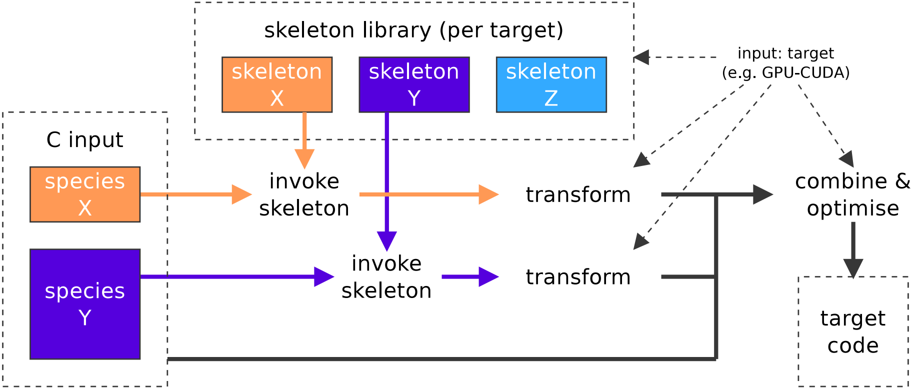
Introducing Bones (1/2)

BONES provides a set of skeletons for multiple targets:

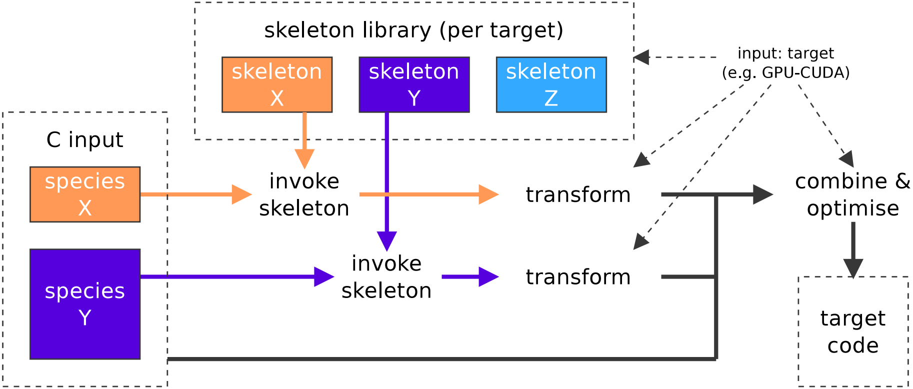
- C-to-CUDA (NVIDIA GPUs)
- C-to-OpenCL (3 targets: AMD GPUs, AMD CPUs, Intel CPUs)
- C-to-OpenMP (multi-core CPUs)
- C-to-C (pass-through)
- C-to-FPGA (under construction)



Introducing Bones (2/2)

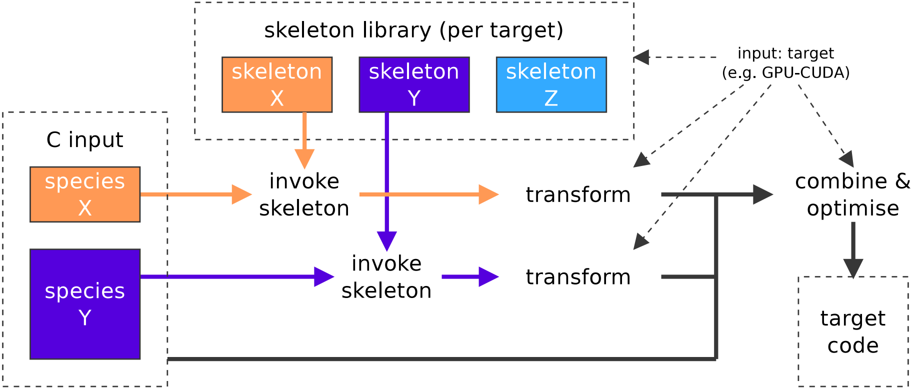


Introducing Bones (2/2)



BONES' main strengths:

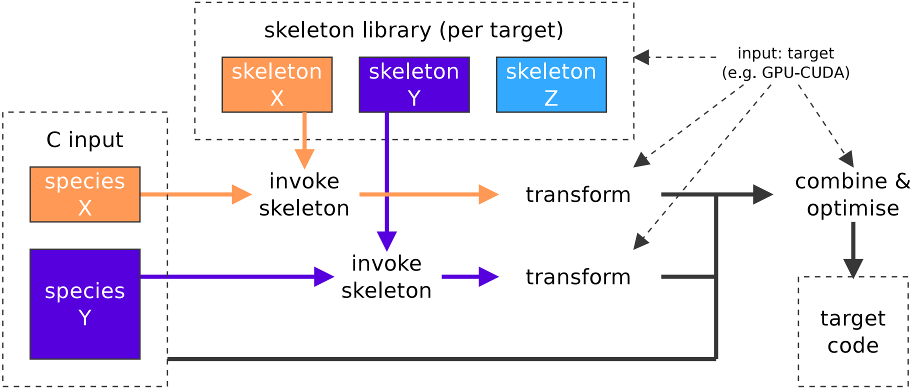
Introducing Bones (2/2)



Bones' main strengths:

1. Code readability: skeletons allow for a lightweight compiler

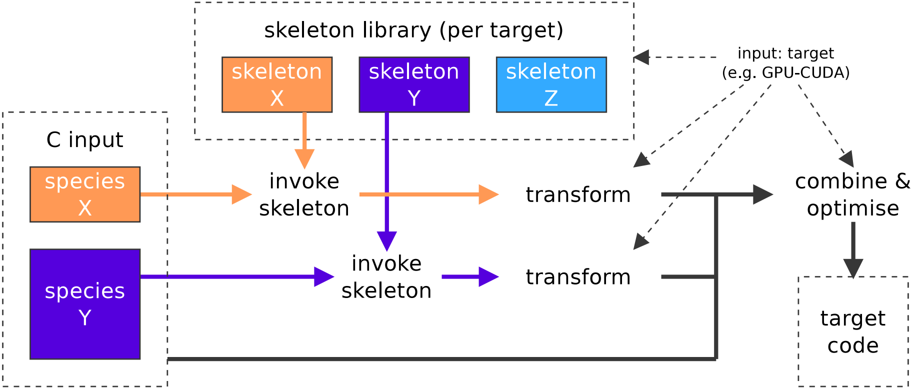
Introducing Bones (2/2)



Bones' main strengths:

1. Code readability: skeletons allow for a lightweight compiler
2. Performance: Write optimised skeletons in the native language

Introducing Bones (2/2)



BONES' main strengths:

1. Code readability: skeletons allow for a lightweight compiler
2. Performance: Write optimised skeletons in the native language
3. Low programmer effort: Species can be extracted automatically

Example simplified skeleton within Bones

Example OpenMP skeleton:

```
int count;
count = omp_get_num_procs();
omp_set_num_threads(count);
#pragma omp parallel
{
  int tid, i;
  int work, start, end;
  tid = omp_get_thread_num();
  work = <parallelism>/count;
  start = tid*work;
  end = (tid+1)*work;
  for(i=start; i<end; i++) {
    <ids>
    <code>



}
```

Example simplified skeleton within Bones

Example OpenMP skeleton:

```
int count;
count = omp_get_num_procs();
omp_set_num_threads(count);
#pragma omp parallel
{
  int tid, i;
  int work, start, end;
  tid = omp_get_thread_num();
  work = <parallelism>/count;
  start = tid*work;
  end = (tid+1)*work;
  for(i=start; i<end; i++) {
    <ids>
    <code>



}
```

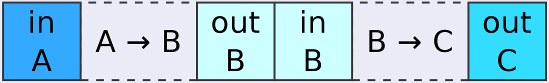
Instantiated skeleton:

```
int count;
count = omp_get_num_procs();
omp_set_num_threads(count);
#pragma omp parallel
{
  int tid, i;
  int work, start, end;
  tid = omp_get_thread_num();
  work = 128/count;
  start = tid*work;
  end = (tid+1)*work;
  for(i=start; i<end; i++) {
    int gid = i;
    r[gid] = 0;
    for (j=0; j<128; j++)
      r[gid] += M[gid][j] * v[j];
  }
}
```

Outline

Example of host-accelerator transfer optimisations

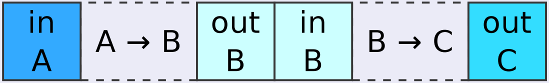
GPU example with 2 kernels:



- Kernel 1: consumes A, produces B
- Kernel 1: consumes B, produces C
- Copy-in before and copy-out directly after the kernel

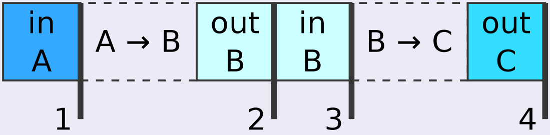
Example of host-accelerator transfer optimisations

GPU example with 2 kernels:



- Kernel 1: consumes A, produces B
- Kernel 1: consumes B, produces C
- Copy-in before and copy-out directly after the kernel

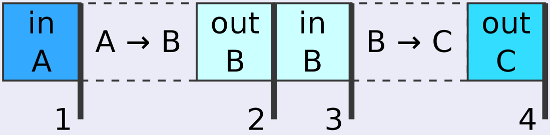
Asynchronous copies:



- Create a second thread to perform asynchronous copies
- Use synchronisation barriers

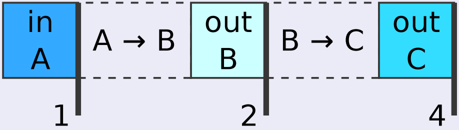
Example of host-accelerator transfer optimisations

Asynchronous copies:



- Create a second thread to perform asynchronous copies
- Use synchronisation barriers

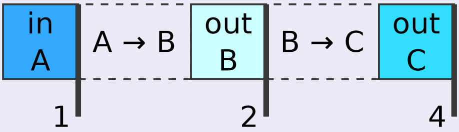
Remove redundant copies:



- Remove the redundant copy-in of B

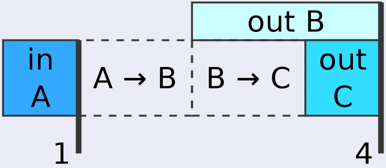
Example of host-accelerator transfer optimisations

Remove redundant copies:



- Remove the redundant copy-in of B

Postpone copy-outs:



- Postpone the copy-out of B
- Overlap transfers with computations

Outline

Performance results for CPUs

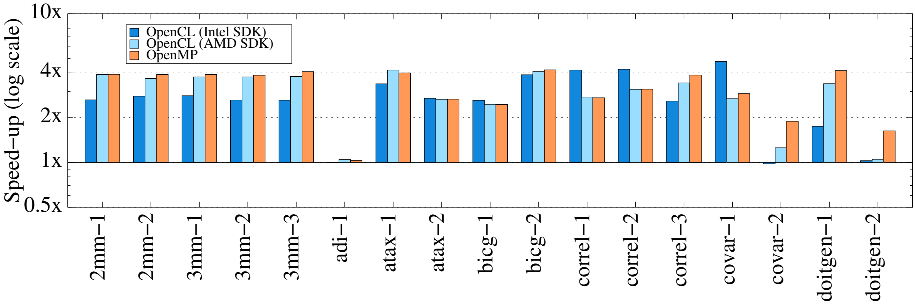
Performance results for execution on a 4-core i7 CPU:

- Based on kernels from the PolyBench benchmark set
- Targets: OpenCL (2 different SDKs) and OpenMP
- Showing speed-ups of individual kernels
- Comparing to unoptimised sequential C-code

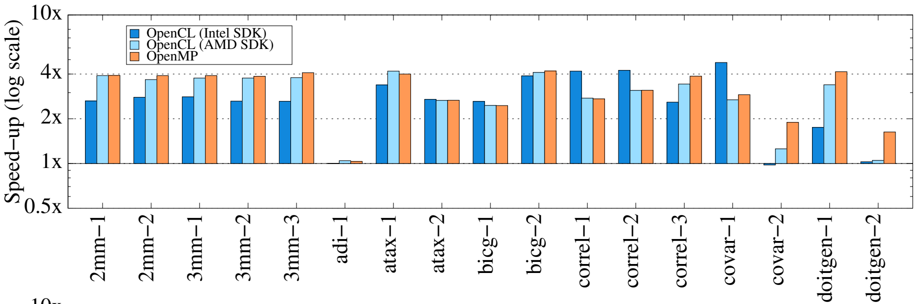
Performance results for CPUs

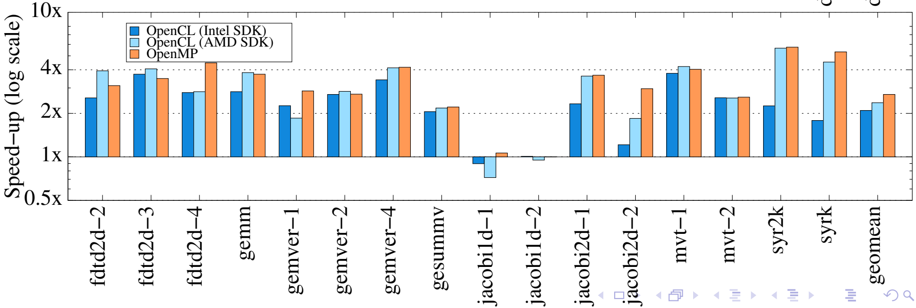
Performance results for execution on a 4-core i7 CPU:

- Based on kernels from the PolyBench benchmark set
- Targets: OpenCL (2 different SDKs) and OpenMP
- Showing speed-ups of individual kernels
- Comparing to unoptimised sequential C-code



Performance results for CPUs





Performance results for GPUs

Comparing BONES to others on a GPU:

- Based on kernels from the PolyBench benchmark set
- Target: CUDA on a GTX470 GPU
- Showing speed-ups of BONES over PAR4ALL and PPCG
  [PAR4ALL and PPCG are the only other automatic C-to-CUDA compilers available]
- Higher is in favour of BONES

Performance results for GPUs

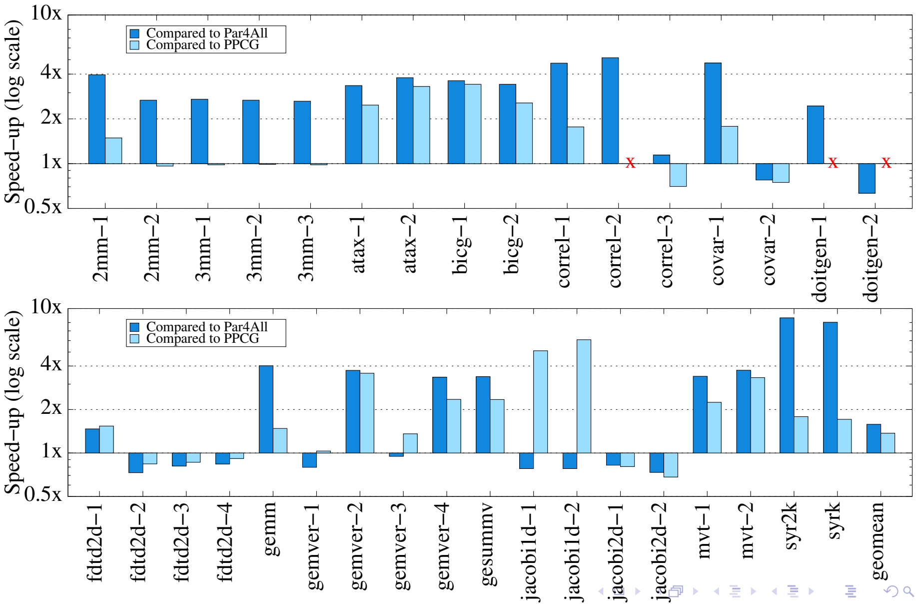
Comparing Bones to others on a GPU:

- Based on kernels from the PolyBench benchmark set
- Target: CUDA on a GTX470 GPU
- Showing speed-ups of Bones over Par4All and PPCG
  [Par4All and PPCG are the only other automatic C-to-CUDA compilers available]
- Higher is in favour of Bones

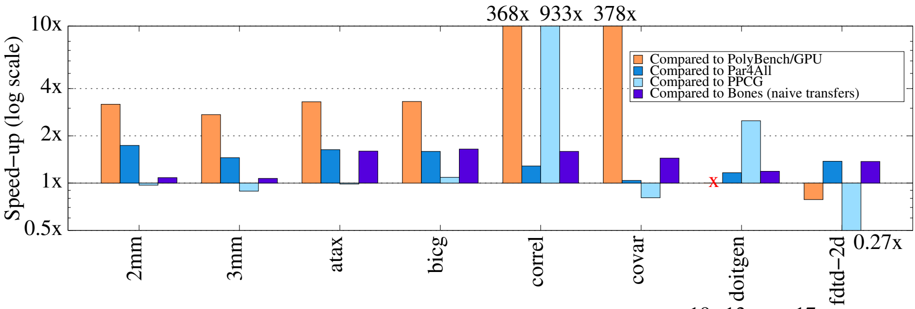
Two different experiments:

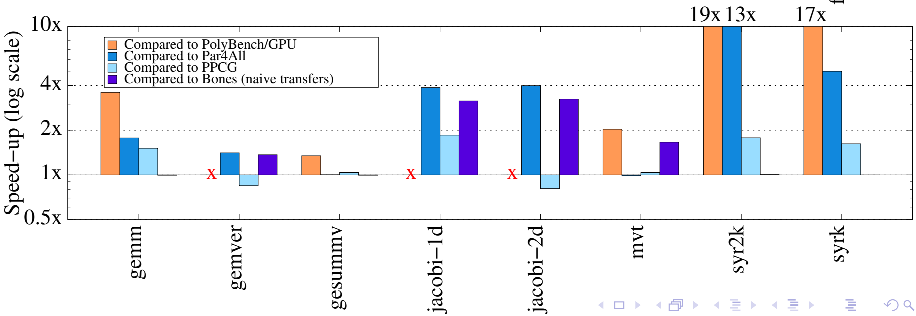
1. Individual GPU kernels
2. Full program, including host-accelerator transfers

Performance results for GPUs (kernel only)



Performance results for GPUs (full program)





Outline

Summary

The source-to-source compiler BONES:

- Uses algorithmic skeletons
- Generates readable CUDA/OpenCL/OpenMP code
- Delivers competitive GPU performance
- Performs host-accelerator transfer optimisations
- Is based on algorithmic species

Summary

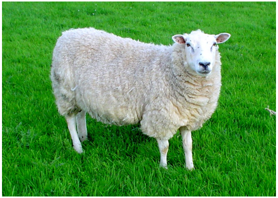
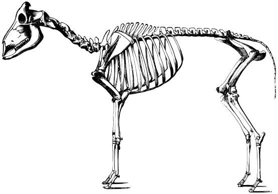
The source-to-source compiler BONES:

- Uses algorithmic skeletons
- Generates readable CUDA/OpenCL/OpenMP code
- Delivers competitive GPU performance
- Performs host-accelerator transfer optimisations
- Is based on algorithmic species

The classification 'algorithmic species':

- Captures memory access patterns from C source code
- Automates classification through ASET and A-DARWIN

Questions / further information

Thank you for your attention!

BONES is available at:
http://parse.ele.tue.nl/bones/

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