GpuGen: Bringing the Power of GPUs into the Haskell World

Sean Lee
seanl@cse.unsw.edu.au

Programming Languages and Systems
School of Computer Science and Engineering
The University of New South Wales
In affiliation with NVIDIA Corporation

02 Sep 2008

Graphics Processing Units (GPUs)

GPUs

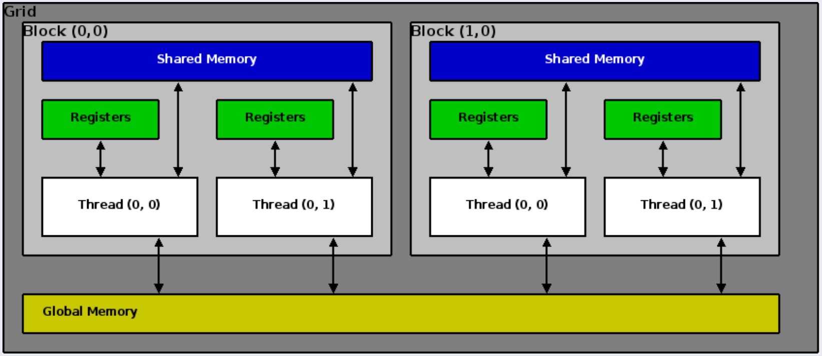
- Superior performance / cost ratio compared to CPUs
- Programmable in SIMD or MIMD style, thus suitable for data parallelism

Compute Unified Device Architecture (CUDA)

- Available on GeForce 8 (G80) and beyond
- A C++ variant designed for *General Purpose GPU computing* (GPGPU)
- Hardware-based program decomposition:
  Functions to run on GPUs (kernels) are prefixed with `__global__` or `__device__`.
- Specialised memory hierarchy and thread hierarchy

Compute Unified Device Architecture (CUDA)

Structure



Compute Unified Device Architecture (CUDA)

Block

- No synchronization across multiple blocks
- Limited number of registers per block
- `blockIdx`

Warp

- A group of threads that run physically in parallel
- `warpSize`

Thread

- Synchronisation within a block
- Shared memory access within a block
- `threadIdx`, `blockDim`

Global Memory and Memory Coalescing

Global Memory

- Global memory access is one of the heaviest overhead in CUDA (400 - 600 cycles)
- Global memory accesses within a half-warp must be coalesced to minimise the number of memory transactions.
- The difference could be an order of magnitude or more.

Memory Coalescing Rules

- The start address of the block has to be aligned to the multiple of region size.
- The *k*th thread in the half-warp must access *k*th element in the block that is being read, or not participate in the memory access.

Shared Memory and Bank Conflicts

Shared Memory

- The shared memory access is about hundreds of times faster than global memory access.
- The shared memory could be used as the software cache to minimise the global memory accesses or to coalesce the global memory accesses.
- The shared memory is split into equally-sized banks.

Bank Conflicts Rules

- If more than one threads try to access the elements in the same bank, a bank conflict occurs and those accesses are serialised.
- If all threads in a half-warp read from an address within the same 32-bit word, there is no bank conflict.

CUDA

Programming in CUDA

- Hardware-specific knowledge is required.
- Algorithms must be re-written to fit in the architecture.
- The optimal configuration changes depending on the algorithms.

Haskell + CUDA

What if Haskell could provide a high-level abstraction that

- takes care of those low-level details,
- lets you focus on *WHAT to program* rather than *HOW to program*, and
- provides you with a similar performance?

Haskell + CUDA

What if Haskell could provide a high-level abstraction that

- takes care of those low-level details,
- lets you focus on *WHAT to program* rather than *HOW to program*, and
- provides you with a similar performance?

GPGPU community

GPU programming becomes easier!

Haskell community

Free performance gain!

GpuGen

How can we combine Haskell and CUDA?

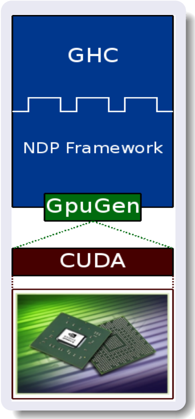
GpuGen

How can we combine Haskell and CUDA? GpuGen

Overview

GpuGen

- A compiler project in progress to compile Haskell to CUDA.
- Takes the external representation of Haskell's intermediate language, *Core*, as the input, and generates the corresponding optimised CUDA code.
- Will be plugged into *Nested Data Parallelism* (NDP) framework in *Glasgow Haskell Compiler* (GHC).



Overview (Cont'd)

Which subset of the language is to be compiled?

The collective list operations are to be compiled:

- Most loops can be expressed with one or more collective list operations.
- The collective list operations well-fit into *Single Instruction, Multiple Data* (SIMD) and *Multiple Insturction stream, Multiple Data* (MIMD) models.
- Unsegmented and segmented `map`, `scan`, `fold`, `filter`, `zipWith`, `permute`, `bpermute`, and many more.

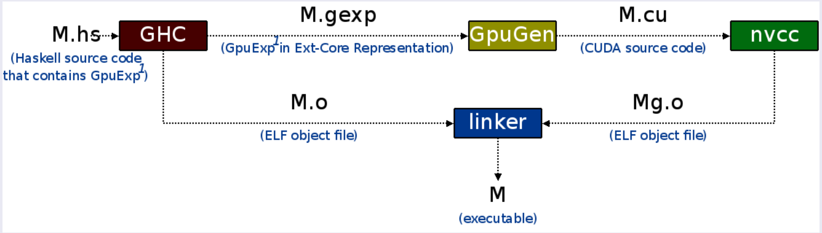
Overview (Cont'd)

Benefits

- Expression-level kernel extraction, resulting in functional program-decomposition (not hardware-based)
- Isolation of the side-effects
- Nested data processing (Flattening and Fusion)
  `[[1, 5, 2, 3, 5], [], [13]]`
- *What*, not *How*

Overview (Cont'd)

Design



1

[1] GpuExp is a *Generalised Algebraic Data Type* (GADT) to describe the operations on GPU.

GpuExp

GpuExp

- a GADT to describe what operations are to run on GPU.
- an *Embedded Domain Specific Language* (eDSL).

GpuExp

GpuExp

- a GADT to describe what operations are to run on GPU.
- an *Embedded Domain Specific Language* (eDSL).

Saxpy in plain Haskell

```
xs, ys ::  [Float] -- type signature
zipWith (\x y -> 5 * x + y) xs ys
```

Saxpy in GpuExp

```
xs, ys ::  Arr CInt (ForeignPtr CFloat)
run (ZipWith (\x y -> 5 * x + y) xs ys)
```

GpuExp (Cont'd)

GpuGen-generated Saxpy CUDA code

```
__device__ float _zipWithOpHSaxpy(float x, float y)
{
  return 5 * x + y;
}
```

GpuExp (Cont'd)

GpuGen-generated Saxpy CUDA code

```
__global__ void _zipWithHSaxpy
  (const float *d_in0, const float *d_in1, float *d_out, uint n, uint isFullBlock)
{
                                  ...
  __syncthreads();
  i = aiDev * 4;
  tempData0 = inData0[aiDev];
  tempData1 = inData1[aiDev];
  if (isFullBlock || i + 3 < n) {
    tempData.x = _zipWithOpHSaxpy(tempData0.x, tempData1.x);
    tempData.y = _zipWithOpHSaxpy(tempData0.y, tempData1.y);
    tempData.z = _zipWithOpHSaxpy(tempData0.z, tempData1.z);
    tempData.w = _zipWithOpHSaxpy(tempData0.w, tempData1.w);
    outData[aiDev] = tempData;
  } else {
    if (i < n) {
      d_out[i] = _zipWithOpHSaxpy(tempData0.x, tempData1.x);
      if (i + 1 < n) {
        d_out[i + 1] = _zipWithOpHSaxpy(tempData0.y, tempData1.y);
        if (i + 2 < n) {
          d_out[i + 2] = _zipWithOpHSaxpy(tempData0.z, tempData1.z);
        }
      }
    }
  }
                                  ...
}
```

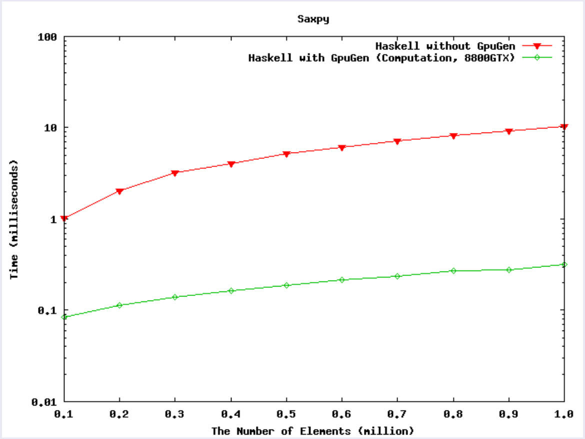
GpuExp (Cont'd)

GpuGen-generated Saxpy CUDA code

```
void gpuZipWithHSaxpy(float *xs0, float *xs1, float *res, uint n)
{
                                  ...
  CUDA_SAFE_CALL(cudaMalloc((void **) &d_xs0, memSize0));
  CUDA_SAFE_CALL(cudaMalloc((void **) &d_xs1, memSize1));
  CUDA_SAFE_CALL(cudaMalloc((void **) &d_res, memSize));
  CUDA_SAFE_CALL(cudaMemcpy(d_xs0, xs0, memSize0, cudaMemcpyHostToDevice));
  CUDA_SAFE_CALL(cudaMemcpy(d_xs1, xs1, memSize1, cudaMemcpyHostToDevice));
  CUT_CHECK_ERROR("Kernel execution failed");
  numBlocks = max(1, (uint) ceil((double) n / 512.0));
  _zipWithHSaxpy<<<dim3 (numBlocks), dim3 (64)>>>(d_xs0, d_xs1, d_res, n, n == numBlocks * 512);
  CUT_CHECK_ERROR("Kernel execution failed");
  CUDA_SAFE_CALL(cudaMemcpy(res, d_res, memSize, cudaMemcpyDeviceToHost));
  CUDA_SAFE_CALL(cudaFree(d_xs0));
  CUDA_SAFE_CALL(cudaFree(d_xs1));
  CUDA_SAFE_CALL(cudaFree(d_res));
}
```

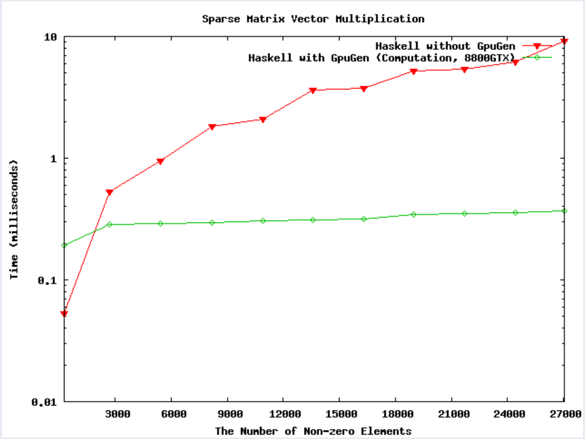
Examples

Saxpy



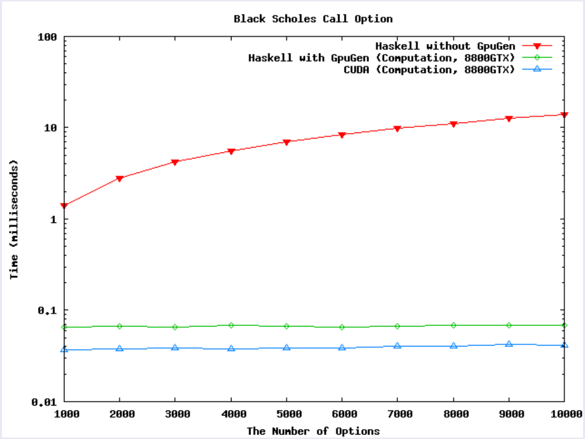
Examples (Cont'd)

Sparse Matrix Vector Multiplication



Examples (Cont'd)

Black Scholes Call Option



Current Status

Towrads the first release

- GpuExp is in progress.
- Core parser is done.
- Translation of Core to CUDA is in progress.
- CUDA code generation for the collective list operations is done. (More operations could be added at any time.)

GpuGen

Fusion

Minimises

- the data trasnfer between the host memory and the device memory, and
- the number of loops.

e.g. `map` + `scanl` = `scanl`

GpuGen

Fusion

Minimises

- the data trasnfer between the host memory and the device memory, and
- the number of loops.

e.g. `map` + `scanl` = `scanl`

Recursion

- Function pointers?

GpuGen

Fusion

Minimises

- the data trasnfer between the host memory and the device memory, and
- the number of loops.

e.g. `map` + `scanl` = `scanl`

Recursion

- Function pointers?

CodeGen to PTX

- Haskell Core $\longrightarrow$ PTX
- More room for optimisations