GPU programming in Haskell

Henning Thielemann

2015-01-23

Tetravue

`http://tetravue.com/`

- 3d camcorder
- not just RGB images, but RGBZ (Z = depth)

Sensor calibration

my task:

- determine correction function for measured depths for every sensor
- more than a million sensors
- 1s per sensor $\sim$ 12 days whole camera calibration
  0.1s per sensor $\sim$ 28h whole camera calibration
  0.01s per sensor $\sim$ 3h whole camera calibration

my favorite implementation language:

- Haskell

First approach to calibration: computation on CPU

Hmatrix

- linear algebra
- rich high-level functions out of the box
- based on LAPACK/BLAS
  - internally uses vector computing
  - internally processes objects in cache-friendly chunks
- works with many GHC (Haskell compiler) versions
- first application prototype: two weeks
- adaption to changed requirements (saturated measurements): two weeks

Second approach: use graphics processor (GPU)

- Graphic processors evolved
  from accelerators for special graphic operations
  to general purpose massive parallel processors.
- GPU less flexible than CPU, but more computing power
- "GPGPU"
  (General-purpose computing on graphics processing units)
- calibration perfectly fits to GPU programming scheme

Nvidia GPU programming

CUDA – formerly Compute Unified Device Architecture

- an extended C programming language – how inspiring
- lock-step parallelism
- divide program into small threads
- e.g., one thread per pixel in an image

Haskell GPU support

Program CUDA from Haskell

- `accelerate`: high-level, large range of back-ends
- `Obsidian`: mid-level, small range of back-ends
- `cuda`: low-level – plain bindings to CUDA language

Accelerate back-ends

back-end	addresses	state
Interpreter	testing	works
CUDA	Nvidia graphic cards	works
CL	any graphic card through OpenCL	prototype
LLVM	any processor through LLVM	prototype
Repa	any processor in plain Haskell	stalled
FPGA	programmable hardware	fictional

Second approach to calibration: use GPU

Accelerate-CUDA

pros:

- array programming abstracts from GPU
- no need to learn CUDA and GPU internals

cons:

- need to implement high-level functions already provided by Hmatrix
- type-correct Accelerate programs may fail at runtime due to missing implementations in CUDA back-end
- Accelerate always needs cutting-edge Haskell compiler GHC
- problematic on MS Windows

Second approach to calibration: results

Accelerate-CUDA: effort needed

- learning Accelerate and porting from Hmatrix: two weeks
- however: fails at run-time
- getting it running: one month
- CUDA version 10 times slower than Hmatrix version
- optimizations with CUBLAS and Obsidian: another month
- still slower than Hmatrix

Nvidia advertisement

- CPU:
  - 4 cores
  - keep illusion of a sequential processor from the 80's: microcode, pipelining, simulate registers, execution re-ordering, superscalarity, hyper-threading, cache
  - can run an operating system
- GPU:
  - 96 cores
  - pure computation power
  - needs a supervising system

Reality

- CPU:
  - 8 `float` multiplications per core (AVX vector computing)
  - 2.20 GHz
  - every of 4 cores operates independently
- GPU:
  - 1 `float` multiplication per core
  - 0.95 GHz
  - 96 cores organized as 2 independent processors with 48 cores
  - still needs space for special graphic operations
  - transfer of input and output between CPU and GPU
  - transfer parallel to GPU computing – programming overhead

- $\frac{96 \cdot 1 \cdot 0.95}{4 \cdot 8 \cdot 2.20} \approx 1.3$
- accelerate factors around 100 from CPU to GPU $\rightarrow$ nonsense
- achieved by comparing optimized GPU code
  with non-vectorized CPU programs

Haskell Accelerate framework

pros

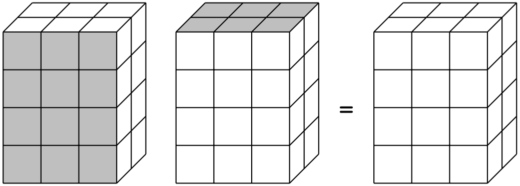
- elegant array programming model
- high-level array transformations instead of low-level loops
  $\rightarrow$ good for programmer and parallelization
- array fusion

cons

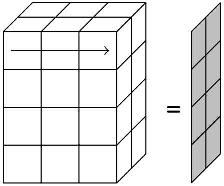
- Embedded Domain Specific Language (EDSL)
- need to rewrite plain Haskell code
- too many problems are only caught at runtime
  e.g. type-correct $\neq$ translatable to compilable CUDA

Example: matrix multiplication 4×3 with 3×2

`zipWith (*)`



`fold1 (+)`



Example: matrix multiplication

```
type Matrix ix a = A.Acc (A.Array (ix:.Int:.Int) a)

multiplyMatrixMatrix ::
   (A.Shape ix, A.Slice ix, A.IsNum a, A.Elt a) =>
   Matrix ix a -> Matrix ix a -> Matrix ix a
multiplyMatrixMatrix x y =
  case (matrixShape x, matrixShape y) of
    (_ :. rows :. _cols, _ :. _rows :. cols) ->
      A.fold1 (+) $ transpose $
      A.zipWith (*)
        (A.replicate (A.lift $ Any:.All:.All:.cols) x)
        (A.replicate (A.lift $ Any:.rows:.All:.All) y)
```

- `replicate`, `zip`, `fold` instead of loops
- relies on array fusion
- one implementation for single and batched operation

→ much more fundamental and elegant than MatLab

MatLab vs. Accelerate

MatLab (proprietary) / Octave (free clone)

- used by many scientists and engineers for numerical computations
- for building prototypes and eternal prototypes :-)
- typing discipline: (almost) everything is a complex valued array
- praised for loop-less programming
- problem:
  no general scheme for loop-less programming like map/reduce, only fixed operations like vector valued addition, dot product and `cumsum`

MatLab: manual matrix multiplication

```
function C = matmul(A,B)
   [ra,ca] = size(A);
   [rb,cb] = size(B);
   C = zeros(ra,cb);
   for k = 1:ra
      for j = 1:cb
         C(k,j) = dot(A(k,:), B(:,j));
      end
   end
```

- loop-less dot product
- still two loops required
- → more difficult to compute parallelly
- → more bound-checking

MatLab: batched matrix multiplication

```
function C = matmul_batched(A,B)
    [na,ra,ca] = size(A);
    [nb,rb,cb] = size(B);
    n = min(na,nb);
    C = zeros(n,ra,cb);
    for k = 1:n
        C(k,:,:) =
            reshape(A(k,:,:),ra,ca) *
            reshape(B(k,:,:),rb,cb);
    end
```

- one loop required
- different implementations for single and batched operation

Accelerate-CUDA: Matrix multiplication performance

- 5-8 times of Hmatrix time on a single CPU core,
  10 times of CUBLAS time (`gemmBatched`)
- Nvidia's profiler hardly useful in connection with Accelerate
- suspicion: not much use of "Shared Memory"
  (kind of explicit cache)
  as proposed by CUDA programming guide

"quick" solution:

- CUBLAS (however, in calibration other slow parts remain)
- requires initialization, contradicts functional approach

Accelerate-CUDA problems

runtime failures

- non-closed functions in `awhile` (now fixed)
- `divMod` not implemented (now fixed)
- operation not supported by back-end (should be type error)
- nested data-parallelism possible in Accelerate language
  - only flat data-parallelism possible on GPU,
    not enforced by type-system
  - problem 1: free usage of array indexing `(!)`
  - problem 2: conversion scalar expression $\leftrightarrow$ singleton array
- GPU launch time-out
  - strange pipeline operator `>->` for breaking fusion
  - more hack than solution

type failures

- `Complex` is not `IsNum`
  broken type class hierarchy using `FlexibleInstances`
- no custom `Array` types possible

Obsidian

- mid-level programming of CUDA, OpenCL and sequential C on CPU
- explicit control of parallelism arrangement in Threads, Thread blocks, Grid
- supports batched monadic/imperative programming

my applications:

- Cholesky decomposition for band-matrices:
  based on `mapAccum` (not available in Accelerate)
- pivot vector to permutation array conversion:
  requires mutable manipulation (not complete in Obsidian)
- call Obsidian code from Accelerate

Patch image

goal:

- compose big image from multiple flat scans
- more restricted but more accurate than panorama stitchers like Hugin

processing steps:

- orientate horizontally
- find positions using CUFFT Fourier transform
- merge parts smoothly

problems with Accelerate-CUDA:

- `Complex` not instance of `IsNum`
- launch time-outs
- too slow

Conclusion

Getting full computation power:

- high performance – not only multi-core
- mind vector computing: Neon; AltiVec; MMX, SSE, AVX
- mind cache locality

GPUs:

- GPU power much less than advertised
- time needed to port program to GPU
- time needed to maintain both CPU and GPU version
- GPU-like parallelism possible with vectors on CPU, too

Conclusion

If someone claims high acceleration factors when porting code from CPU to GPU, ask him whether he optimized his CPU code by

- vector computing
- cache friendly memory access patterns

Conclusion

Haskell:

- elegant GPU computing through Accelerate
- performance may be bad
    - failed fusion
    - expensive memory access patterns
    - no control over shared memory (= explicit cache)
- current performance makes it useless
- better use Hmatrix for linear algebra for now
- NVBLAS even moves Hmatrix computations to GPU

Conclusion

various restrictions by several parts:

- vendor lock-in to Nvidia's CUDA framework and libraries (free of charge but closed-source)
- update to new CUDA version removes support for older GPUs
- GPU requires lock-step parallelism
- Accelerate: immutable operations, no batched `mapAccum`/`scan`
- Obsidian: batched `mapAccum`, may support mutable manipulations someday

Final Conclusion

- not enough to move computation from CPU to GPU
- weakest link in the chain:
  one slow Accelerate operation can make the whole GPU programming useless