Accelerated Protein matching, Using GPUs

Trevor L. McDonell
University of New South Wales

Case Study: Protein Matching

- Still difficult to write parallel software
- Parallel implementation of a real world algorithm
  - use common Haskell functions, implemented on the GPU
  - hide GPU architectural details

SEQUEST

A A I K G K I D V C I V L L Q H K A E P T I R N T D G R T A

Illustration by Tony Boudreault

SEQUEST

AAIK

GK

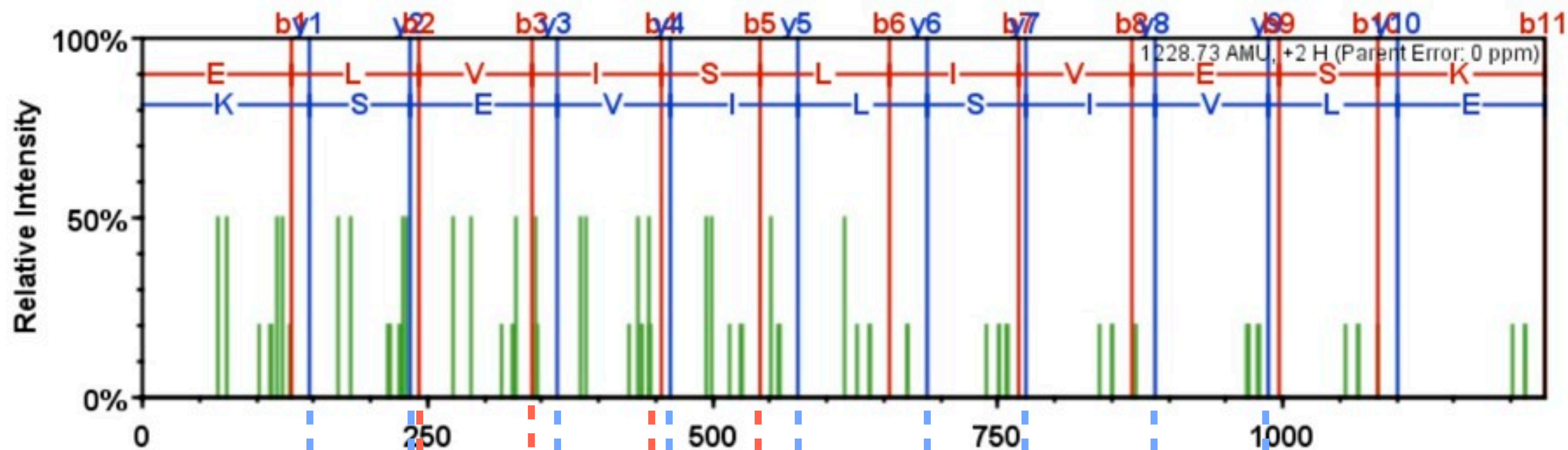
IDVCIVLLQHK

AEPTIR

NTDGR

TA

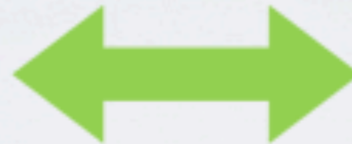
Illustration by Tony Boudreault



b2 b3 y4 y5 y6 y7 y8 y9
1228.73 AMU, +2 H (Parent Error: 3.4 ppm)
E L V I S L I V E S K
K S E V I L S I V L E
Relative Intensity
100%
50%
0%
b4-H2O
b3-H2O
a2
y3
y2
y1
b1
y8+2H
b5-H2O
b6-H2O
b4
b4-NH3
y9+2H
b5
b6
b7
b8
b9
b10
y10
0
250
500
750
1000
M/z

Quadcore
Xeon CPU

Tesla T10 GPU



- Tens of thousands of data parallel threads
- Speedups of 10x to 200x have been observed in real applications

import Foreign.C

```
mallocBytes :: Int -> IO (Ptr a)
free :: Ptr a -> IO ()

newArray :: [a] -> IO (Ptr a)
withArray :: [a] -> (Ptr a -> IO b) -> IO b
```

import Foreign.CUDA

```
type DevicePtr a

mallocBytes :: Int -> IO (DevicePtr a)
free :: DevicePtr a -> IO ()

newArray :: [a] -> IO (DevicePtr a)
withArray :: [a] -> (DevicePtr a -> IO b) -> IO b
```

Parallel Operations

- Efficient implementations in CUDA, called from Haskell
- Use templates to express type and (associative) operation
- map, zipWith, prescan, permute, backpermute

```
int fold_plusi(int xs, int N) {
  return fold< Plus<int>, int >(xs, N);
}
```

Parallel Operations

Dot Product

```
dotp :: [Float] -> [Float] -> IO Float
dotp xs ys =
    withArray xs      $ \xs'      ->
    withArrayLen ys $ \len ys' ->
    allocaBytes (len * sizeOf (undefined::Float)) $ \zs' -> do
        zipWith_timesf xs' ys' zs' len
        fold_plusf zs' len
```

Results

- Database ~20k proteins

CPU Version	3 seconds

A First Attempt

- Just compute the final score (dot product) on the GPU
- Minimal code changes, just need to deal with IO now

Results

- Database ~20k proteins

CPU Version	3 sec
First version: individual dot product	7 sec

A Second Attempt

- Individual dot products weren't enough to offset the cost of data transfer
- Each protein in the database is split into many peptides, try to consider them all at once

Results

- Database ~20k proteins

CPU Version	3 sec
First version: individual dot product	7 sec
Second version: matrix vector multiply	40 sec

Hmm...

- Turns out most candidates can be filtered out, so spend a lot of time transferring data that is ultimately not used

- Profiling output:

Marshalling

```
 COST CENTRE                   MODULE                 %time %alloc  ticks     bytes

digestProtein                 Protein                 87.2   95.6   6569 16693476851
seqextract                    Protein                  2.4    1.2    180 204206959
scanr1Seg_plusf               Kernels                  1.8    0.0    138    841209
   ...
```

Computation

Conclusion

- I have found two ways to not parallelise this algorithm
- Status:
  - Bindings to CUDA functions
  - Needs a front-end for a more Haskell-ish interface