GPU Programming

Imperative And Functional Approaches

Bo Joel Svensson

April 16, 2013

The multicore era

The number of cores is rising with each generation of CPU or GPU

- Avoid the power wall:
  Increased frequency - increases power consumption.
- Avoid the Instruction level parallelism (ILP) wall:
  Finding and utilising ILP is getting more and more difficult.
- Avoid the Memory wall:
  Increasing speed gap between memory chips and processors.

Modern processors and Accelerators

- NVIDIA GTX 680: GPU
  8 * Multiprocessor (* 192 compute units, called CUDA cores)
- Intel Xeon Phi: Accelerator
  60 x86 cores
  512 bits wide SIMD (16 floats)
- Intel Ivy Bridge: Heterogeneous CPU-GPU
  2 - 6 x86 CPU cores
  6 or 16 GPU "execution units"

New ways of programming

- GPUs:
  - Accelerate: Embedded in Haskell [4].
  - CUDA: NVIDIA's C dialect for their GPUs [1].
  - Nikola: Embedded in Haskell [6].
  - Obsidian: Embedded in Haskell [5].
  - OpenCL: A platform independent "CUDA".
  - Thrust: C++ library [9].

New ways of programming

- Intel Xeon Phi:
  - Use old tools such as: OpenMP, OpenMPI.

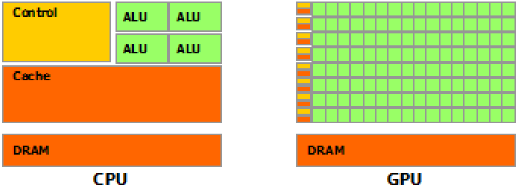
New ways of programming

- Heterogeneous:
  - ArBB: Intel Array building blocks [8]
  - EmbArBB: Haskell embedding of ArBB [11]
  - OpenCL: Can be compiled for both CPU and GPU.
  - Language combinations: CUDA + MPI.

GPUs: The device

- Memory
- Computation
- Cooling
- Connection to Host

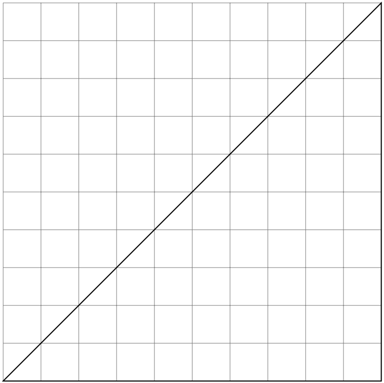
GPUs: Compared to CPUs



[1]

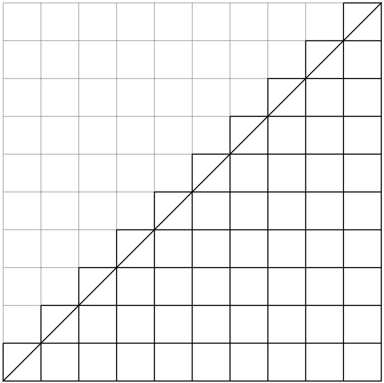
[1] image source:
http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

GPUs: The graphics roots



Built for drawing triangles.

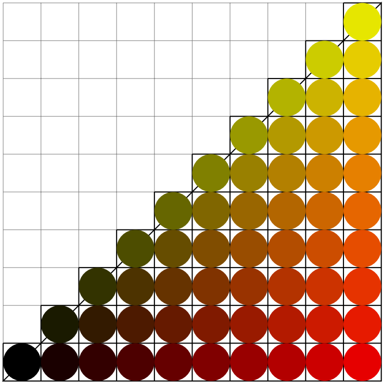
GPUs: The graphics roots



Built for drawing triangles.

For each pixel, a small program is executed.

GPUs: The graphics roots

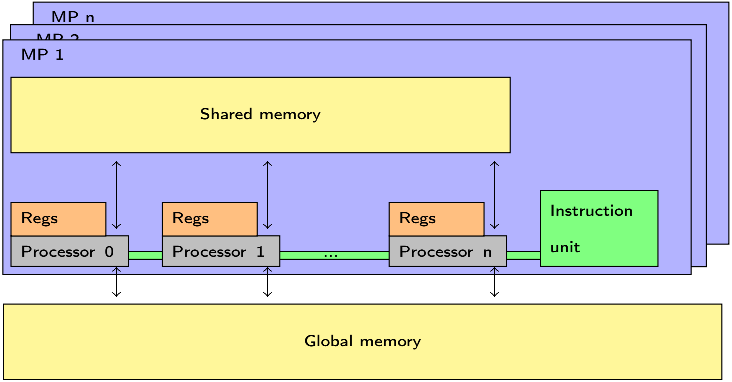


Built for drawing triangles.

For each pixel, a small program is executed.

Each pixel value is computed independently.

CUDA and the GPU: Multiprocessors



CUDA and the GPU: CUDA programming model

- Single Program Multiple Threads SIMT.
- Threads are divided into *Blocks*
  - Up to 1024 threads per block. (Varies!)
  - Many blocks share a MP.
  - Threads within a block can communicate using shared memory.
  - More threads per block than processors per MP
    `__syncthreads();`

CUDA and the GPU: CUDA programming example

The Kernel:

```
__global__ void inc(float *i, float *r){

  unsigned int ix = blockIdx.x * blockDim.x + threadIdx.x;

  r[ix] = i[ix]+1;
}
```

CUDA and the GPU: CUDA glue code part 1

```
#include <stdio.h>
#include <cuda.h>

#define BLOCK_SIZE 256
#define BLOCKS 1024
#define N (BLOCKS * BLOCK_SIZE)
```

CUDA and the GPU: CUDA glue code part 2

```
int main(){

  float *v, *r;
  float *dv, *dr;

  v = (float*)malloc(N*sizeof(float));
  r = (float*)malloc(N*sizeof(float));

  //generate input data
  for (int i = 0; i < N; ++i) {
    v[i] = (float)(rand () % 1000) / 1000.0;
  }
  /* Continues on next slide */
```

CUDA and the GPU: CUDA glue code part 3

```
cudaMalloc((void**)&dv, sizeof(float) * N );
cudaMalloc((void**)&dr, sizeof(float) * N );

cudaMemcpy(dv, v, sizeof(float) * N,
           cudaMemcpyHostToDevice);
/* Continues on next slide */
```

CUDA and the GPU: CUDA glue code part 4

```
inc<<<BLOCKS, BLOCK_SIZE,0>>>(dv,dr);

cudaMemcpy(r, dr, sizeof(float) * N , cudaMemcpyDeviceToHost);

cudaFree(dv);
cudaFree(dr);
/* Continues on next slide */
```

CUDA and the GPU: CUDA glue code part 5

```
  for (int i = 0; i < N; ++i) {
    printf("%f ", r[i]);
  }
  printf("\n");

  free(v);
  free(r);
}
```

CUDA and the GPU: CUDA timing code

Add timing:

```
cudaEvent_t start,stop;

cudaEventCreate(&start);
cudaEventCreate(&stop);
```

CUDA and the GPU: CUDA timing code

Add timing:

```
cudaEvent_t start,stop;

cudaEventCreate(&start);
cudaEventCreate(&stop);
```

Wrap what you want to time with:

```
cudaEventRecord(start);
 /* This part is timed */
cudaEventRecord(stop);
cudaEventSynchronize(stop);
```

CUDA and the GPU: CUDA timing code

```
float elapsedTime;
cudaEventElapsedTime(&elapsedTime, start, stop);

printf("Time %f ms\n", elapsedTime);
```

CUDA and the GPU: Compile and run

```
> nvcc inc.cu -o inc
>./inc

1.383000 1.886000 1.777000 ...
Time 0.022432 ms
```

Haskell approaches to GPU programming

- Accelerate (Data.Array.Accelerate)
- Nikola
- Obsidian

Accelerate programming

Shapes:

```
type Dim0
type Dim1
...
```

Arrays:

```
Array sh e
```

Aliases:

```
type Scalar a = Array Dim0 a
type Vector a = Array Dim1 a
```

Accelerate programming: Example

```
dotp :: Vector Float -> Vector Float -> Acc (Scalar Float)
dotp xs ys = let xs' = use xs
                 ys' = use ys
             in fold (+) 0 (zipWith (*) xs' ys')
```

Accelerate programming: Interface

```
use :: Array sh e -> Acc (Array sh e)
map :: (Exp a -> Exp b)
    -> Acc (Array sh a)
    -> Acc (Array sh b)
zipWith :: (Exp a -> Exp b -> Exp c)
        -> Acc (Array sh a)
        -> Acc (Array sh b)
        -> Acc (Array sh c)
fold :: (Exp a -> Exp a -> Exp a)
     -> Exp a
     -> Acc (Array sh:.Int a)
     -> Acc (Array sh a)
```

And many more: `permute`, `scan` ...

Accelerate programming: More information

Ref. [4] provides more details about Accelerate and its programming model.
Recent progress in optimising accelerate programs is described in Ref. [7]. Here they mention fusion of kernels.

Nikola

- `map`
- `zipWith`

Accelerate & Nikola

- High level languages.
- Relies on code generator to produce good GPU code.

Accelerate & Nikola

- High level languages.
- Relies on code generator to produce good GPU code.
- Obsidian is slightly different...

Obsidian

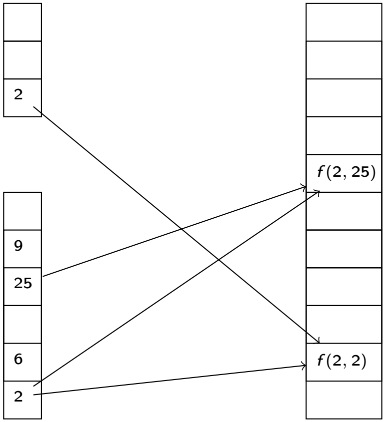
2 3

[2]game: `www.minecraft.net`
[3]lava: An embedded language for hardware design [2]

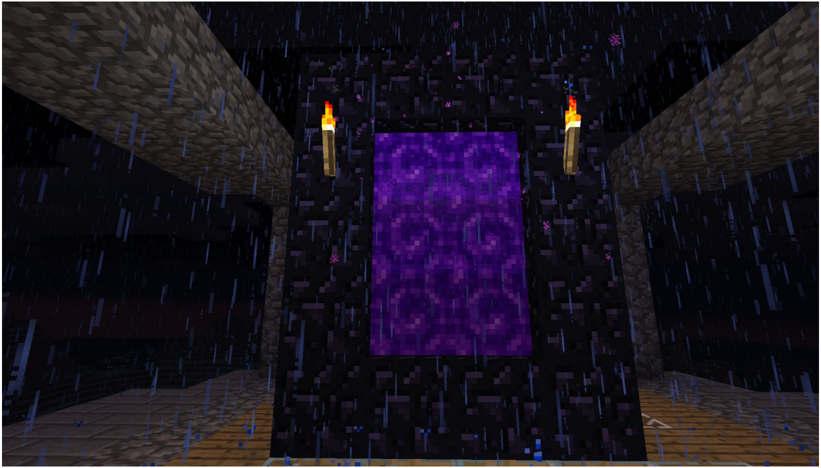
Obsidian: Goals

- Encourage experimentation.
- Combinators for parallel programming.
- Lower level than Accelerate and Nikola.
  - Programmer needs to know a little about GPUs.
  - Programmer control over how to compute on the GPU.
  - Programmer control over shared memory.
- Higher level than CUDA.
  - Not as much indexing *magic*.
  - Program describes “whole” computation.

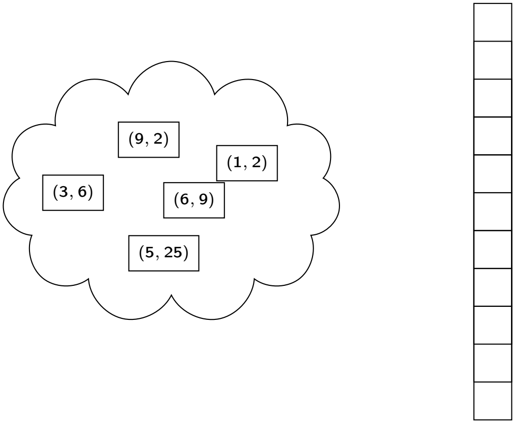
Obsidian: Pull arrays



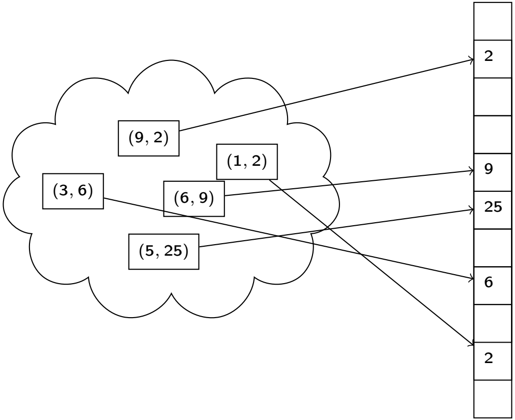
Obsidian: Push arrays



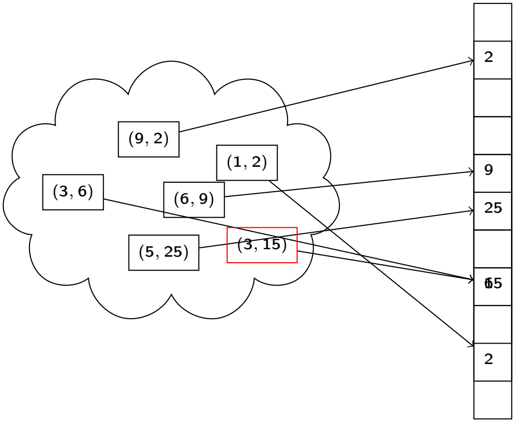
Obsidian: Push arrays



Obsidian: Push arrays



Obsidian: Push arrays



Obsidian: Example kernel

```
inc :: SPull EFloat -> SPull EFloat
inc = fmap (+1)
```

Obsidian: Example grid

```
incG :: DPull EFloat -> DPush Grid EFloat
incG arr = mapG (return . inc) (splitUp 256 arr)
```

Obsidian: Example grid

```
incG :: DPull EFloat -> DPush Grid EFloat
incG arr = mapG (return . inc) (splitUp 256 arr)

splitUp :: Word32 -> DPull a -> DPull (SPull a)
splitUp n arr =
  mkPullArray (m `div` fromIntegral n) $ \i ->
    mkPullArray n $ \j -> arr ! (i * fromIntegral n + j)
  where
    m = len arr
```

Obsidian: Example code generation

```
input :: DPull EFloat
input = namedGlobal "apa" (variable "X")

getIncG = putStrLn $ genKernel "inc" incG input
```

Obsidian: Example code generation

```
input :: DPull EFloat
input = namedGlobal "apa" (variable "X")

getIncG = putStrLn $ genKernel "inc" incG input

> getIncG
__global__ void inc(float* input0,uint32_t n0,float* output0){

    uint32_t t0 = ((blockIdx.x*256)+threadIdx.x);
    output0[t0] = (input0[t0]+1.0);


}
```

Obsidian: Example shared memory

```
incG :: DPull EFloat -> DPush Grid EFloat
incG arr = mapG (force . inc) (splitUp 256 arr)
```

Obsidian: Example shared memory

```
incG :: DPull EFloat -> DPush Grid EFloat
incG arr = mapG (force . inc) (splitUp 256 arr)

> getIncG
__global__ void inc(float* input0,uint32_t n0,float* output0){

    uint32_t t1 = ((blockIdx.x*256)+threadIdx.x);
    extern __shared__ uint8_t sbase[];
    ((float*)sbase)[threadIdx.x] = (input0[t1]+1.0);
    __syncthreads();
    output0[t1] = ((float*)sbase)[threadIdx.x];


}
```

Obsidian: Recap

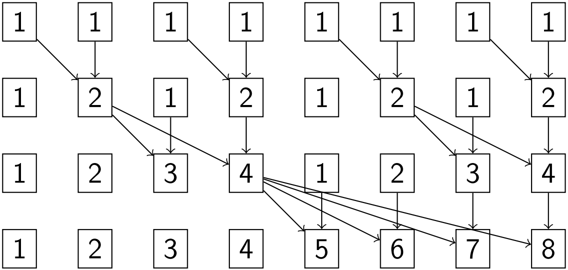
- Arrays: Push and Pull
  - With static (`SPull`, `SPush`) and dynamic (`DPull`, `DPush`) lengths.
- Push arrays have a parameter `Thread`, `Block` or `Grid`.
- ```
  mapG :: ASize l
        =>(SPull a -> BProgram (SPull b))
        -> Pull l (SPull a) -> Push Grid l b
  ```
- ```
  force :: (Pushable p, Array p, MemoryOps a)
         => p Word32 a -> BProgram (Pull Word32 a)
  ```

Obsidian: prefix sums (Scan)

Given a associative operator $\oplus$, and an array $\{a_0, a_1, \cdots, a_{n-1}\}$ the prefix sums operation results in
$\{a_0, (a_0 \oplus a_1), \cdots, a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}\}$

This operation can be implemented on parallel hardware and can be used in the implementation of many algorithms, see Ref. [3] for more info.

Obsidian: Sklansky [10]



Obsidian: Sklansky implementation

```
sklansky :: Int -> SPull EFloat -> BProgram (SPull EFloat)
sklansky 0 arr = return arr
sklansky n arr =
  do
    let arr1 = binSplit (n-1) (fan (+)) arr
    arr2 <- force arr1
    sklansky (n-1) arr2
```

Obsidian: Sklansky implementation

```
sklansky :: Int -> SPull EFloat -> BProgram (SPull EFloat)
sklansky 0 arr = return arr
sklansky n arr =
  do
    let arr1 = binSplit (n-1) (fan (+)) arr
    arr2 <- force arr1
    sklansky (n-1) arr2

fan :: Choice a => (a -> a -> a) -> SPull a -> SPull a
fan op arr = a1 `conc` fmap (op (last a1)) a2
  where (a1,a2) = halve arr
```

Obsidian: Many Sklansky

```
sklanskyG :: DPull EFloat -> DPush Grid EFloat
sklanskyG arr = mapG (sklansky 8) (splitUp 256 arr)
```

Obsidian: Sklansky generated code

```
__global__ void sklansky(float* input0,uint32_t n0,float* output0){

    uint32_t t2 = ((blockIdx.x*32)+((threadIdx.x&4294967294)|(threadIdx.x&1)));
    uint32_t t9 = ((threadIdx.x&4294967292)|(threadIdx.x&3));
    uint32_t t14 = ((threadIdx.x&4294967288)|(threadIdx.x&7));
    uint32_t t19 = ((threadIdx.x&4294967280)|(threadIdx.x&15));
    extern __shared__ __attribute__ ((aligned(16))) uint8_t sbase[];
    ((float*)sbase)[threadIdx.x] =
      (((threadIdx.x&1)<1) ? input0[t2] :
           (input0[((blockIdx.x*32)+((threadIdx.x&4294967294)|0))]+input0[t2]));
    __syncthreads();
    ((float*)(sbase+128))[threadIdx.x] =
      (((threadIdx.x&3)<2) ? ((float*)sbase)[t9] :
           (((float*)sbase)[((threadIdx.x&4294967292)|1)]+((float*)sbase)[t9]));
    __syncthreads();
    ((float*)sbase)[threadIdx.x] =
      (((threadIdx.x&7)<4) ? ((float*)(sbase+128))[t14] :
           (((float*)(sbase+128))[((threadIdx.x&4294967288)|3)]+((float*)(sbase+128))[t14]));
    __syncthreads();
    ((float*)(sbase+128))[threadIdx.x] =
      (((threadIdx.x&15)<8) ? ((float*)sbase)[t19] :
           (((float*)sbase)[((threadIdx.x&4294967280)|7)]+((float*)sbase)[t19]));
    __syncthreads();
    ((float*)sbase)[threadIdx.x] =
      ((threadIdx.x<16) ? ((float*)(sbase+128))[threadIdx.x] :
        (((float*)(sbase+128))[15]+((float*)(sbase+128))[threadIdx.x]));
    __syncthreads();
    output0[((blockIdx.x*32)+threadIdx.x)] = ((float*)sbase)[threadIdx.x];

}
```

Next time

- More about arrays:
  - What is a pull array, really ?
  - What is a push array, really ?
- Programs:
  - `TProgram`
  - `BProgram`
  - `GProgram`
- Implementation:
  - Library functions.
  - Code generation.

End

CUDA programming manual.
`http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh.
Lava: Hardware design in haskell.
In *ICFP*, pages 174–184, 1998.

Guy E. Blelloch.
Prefix sums and their applications.
Technical report, Synthesis of Parallel Algorithms, 1990.

Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover.
Accelerating Haskell array codes with multicore GPUs.
In *Proceedings of the sixth workshop on Declarative aspects of multicore programming*, DAMP '11, New York, NY, USA, 2011. ACM.

Joel Svensson.
Obsidian: GPU Kernel Programming in Haskell.
Technical Report 77L, Computer Science and Enginering, Chalmers University of Technology, Gothenburg, 2011.
Thesis for the degree of Licentiate of Philosophy.

Geoffrey Mainland and Greg Morrisett.
Nikola: embedding compiled GPU functions in Haskell.
SIGPLAN Not., 45(11), 2010.

Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier.
Optimising Purely Functional GPU Programs.
Submitted to ICFP 2013.

Chris J. Newburn, Byoungro So, Zhenying Liu, Michael McCool, Anwar Ghuloum, Stefanus Du Toit, Zhi Gang Wang, Zhao Hui Du, Yongjian Chen, Gansha Wu, Peng Guo, Zhanglin Liu, and Dan Zhang.
Intel's array building blocks: A retargetable, dynamic compiler and embedded language.
In *Proceedings of the 2011 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA, 2011. IEEE Computer Society.

NVIDIA.
NVIDIA Thrust Library.

J. Sklansky.
Conditional Sum Addition Logic.
Trans. IRE, EC-9(2):226–230, June 1960.

Bo Joel Svensson and Mary Sheeran.

Parallel programming in haskell almost for free: an embedding of intel's array building blocks.

In *Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing*, FHPC '12. ACM, 2012.