GPU Programming with Accelerate

Stephen A. Edwards

Columbia University

Fall 2023

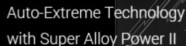
NVIDIA RTX 2060

Example: The NVIDIA Turing TU106 (RTX 2060)

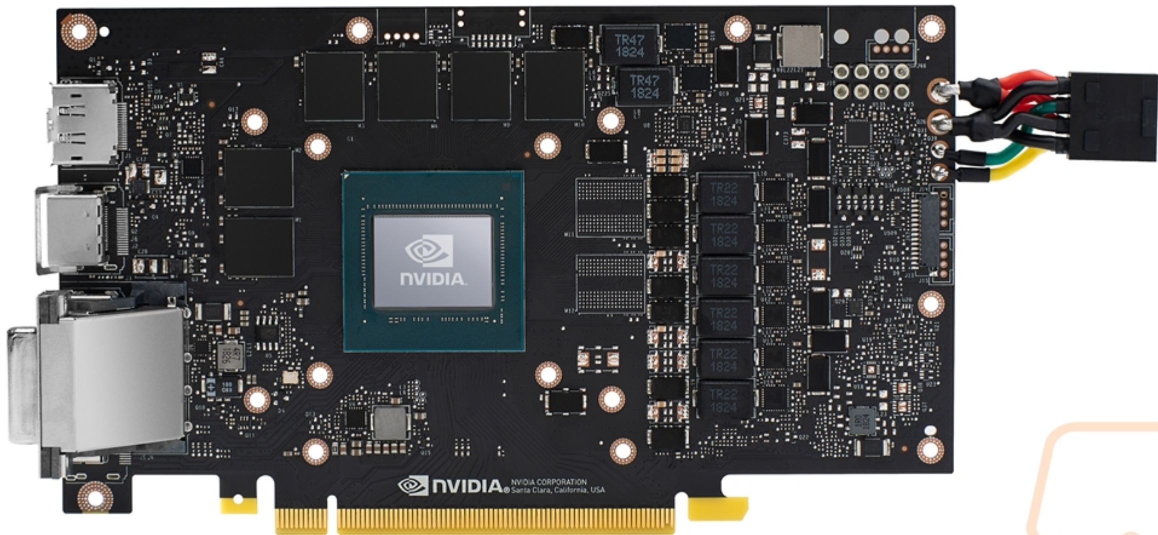
- The Turing Streaming Multiprocessor

The Accelerate Package

- The Array Type: Shapes
- The run function
  - Indexing
- Floyd-Warshall in Accelerate



Aura Sync Lighting Backplate
FanConnect II
Aura RGB Header
Aura Sync
Auto-Extreme Technology with Super Alloy Power II
VR Friendly HDMI ports
A metal frame increases structural integrity
MaxContact Technology
Patented Wing-blade 0dB Fans with IP5X-certified Dust Resistant



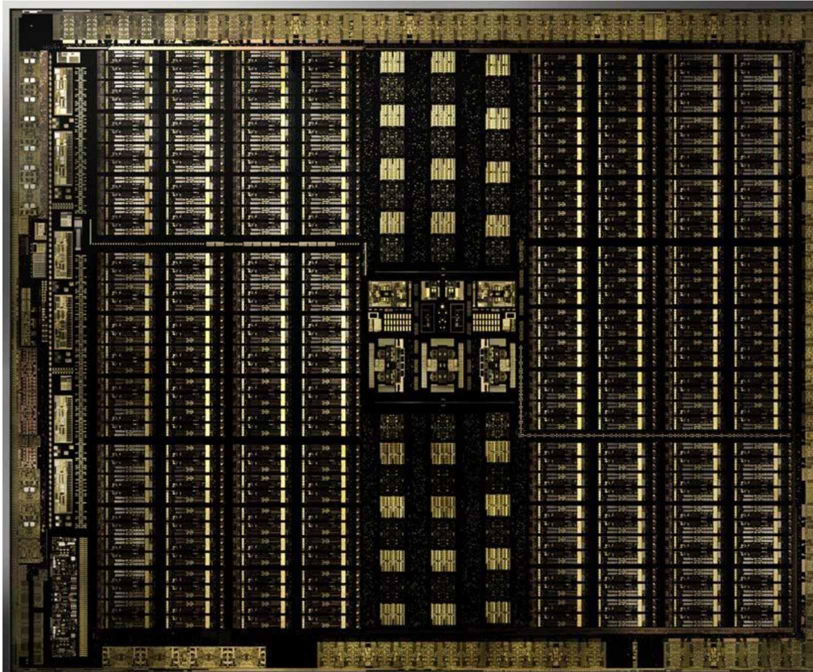
6 GB GDDR6 (space for 8 GB) 192 bits (32/chip) @ 1.75 GHz = 336 Gb/s

NVIDIA Turing TU106-200A

10.8G trans., 12 nm, 445 mm^2

- 3 Graphics Processing Clusters
  - 5 Texture Processing Clusters
    - 2 Streaming Multiprocessors
      - 64 CUDA Cores
      - 256 KB Register file
      - 96 KB L1/shared memory
- 4 MB L2 Cache
- 8 32-bit Memory Controllers

1920 CUDA cores total



My TU106 only enables 5 of 6 TPCs per GPC and 6 of 8 memory controllers

NVIDIA Turing Streaming Multiprocessor

30 on my TU106-200A

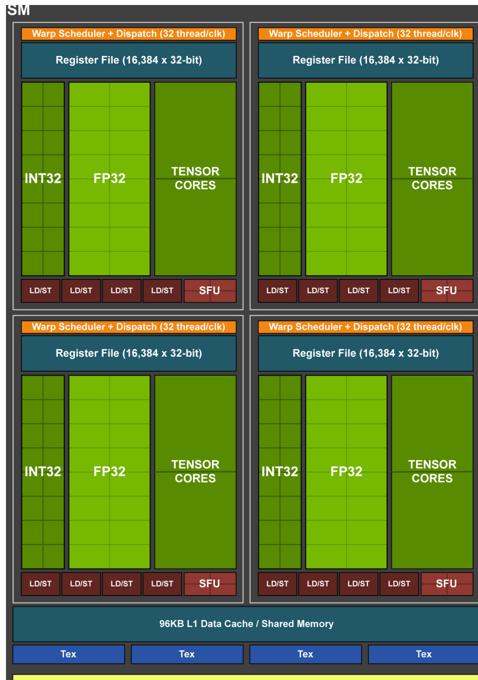
64 + 32 KB L1 Data Cache/shared memory

4 Processing blocks

- 16 FP32 + 16 INT32 (“CUDA”) Cores
- 1 Warp Scheduler
- 1 Dispatch unit
- 64 KB (32 × 16K) Register file

Warp Scheduler + Dispatch unit issues one instruction across *all* 16 CUDA cores

Scheduler manages many (1024?) threads (simultaneous multithreading “hyperthreading”), interleaves their execution based on data availability (from memory)



The Accelerate Package: Parallel Arrays (GPU-accelerated)

Deeply embbedded domain-specific language for array computations

Library compiles your code JIT-like to the target, e.g., GPUs through CUDA

https://hackage.haskell.org/package/accelerate

https://github.com/AccelerateHS/accelerate/

http://developer.nvidia.com/cuda-downloads

apt install cuda-10 nvidia-cuda-toolkit

```
# stack.yaml
extra-deps:
- accelerate-1.3.0.0
- accelerate-llvm-native-1.3.0.0
- accelerate-llvm-ptx-1.3.0.0
- accelerate-llvm-1.3.0.0
- cuda-0.10.2.0
- nvvm-0.10.0.0
```

```
# package.yaml
executables:
  fwaccel:
    ghc-options:
    - -threaded
    - -rtsopts
    - -with-rtsopts=-N
    - -dynamic      # for llvm-native
    dependencies:
    - accelerate
    - accelerate-llvm-ptx # CUDA GPU
    - accelerate-llvm-native # LLVM
```

Import the Accelerate Package

```
import Data.Array.Accelerate as A              -- Construct computations
import Data.Array.Accelerate.Interpreter -- Run them locally
```

or

```
import Data.Array.Accelerate as A         -- Construct computations
import Data.Array.Accelerate.LLVM.PTX -- Run them on the GPU
```

or

```
import Data.Array.Accelerate as A              -- Construct computations
import Data.Array.Accelerate.LLVM.Native -- Using LLVM on the CPU
```

The Array Type: Includes the number of dimensions

Similar to Repa, but no representation argument. Same shape type trick:

```
{-# LANGUAGE TypeOperators #-}

data Array shape elements

data Z             = Z              -- Zero-dimensional: a scalar
data tail :. head = tail :. head -- head: type of the dimension (Int)

type DIM0 = Z             -- Scalar
type DIM1 = DIM0 :. Int -- Vector
type DIM2 = DIM1 :. Int -- 2D Matrix
type DIM3 = DIM2 :. Int -- 3D Array

type Scalar e = Array DIM0 e
type Vector e = Array DIM1 e
type Matrix e = Array DIM2 e
```

First Example

```
*Main> :t fromList
fromList :: (Shape sh, Elt e) => sh -> [e] -> Array sh e

*Main> fromList (Z:.10) [1..10] :: Vector Int
Vector (Z :. 10) [1,2,3,4,5,6,7,8,9,10]

*Main> let arr = fromList (Z:.3:.5) [1..] :: Array DIM2 Int
*Main> arr
Matrix (Z :. 3 :. 5)
  [  1,  2,  3,  4,  5,
     6,  7,  8,  9, 10,
    11, 12, 13, 14, 15]

*Main> :t indexArray
indexArray :: (Shape sh, Elt e) => Array sh e -> sh -> e
*Main> indexArray arr (Z:.2:.1)
12
```

Arrays of tuples

Array elements need to be "simple." Int, Floats, Doubles, and tuples thereof.

```
*Main> fromList (Z:.2:.4) (Prelude.zip [1..] [2,4..])
*Main|                                   :: Matrix (Int, Int)
Matrix (Z :. 2 :. 4)
  [  (1,2),  (2,4),  (3,6),  (4,8),
    (5,10), (6,12), (7,14), (8,16)]
```

Internally, arrays of tuples are turned into tuples of arrays

Running a computation

In general, you need to prepare data on the CPU, download it to the GPU, compile, download, and run your array code there, and then read the results back to the CPU.

```
A.map :: (Shape sh, Elt a, Elt b) -- Shape and element types
      => (Exp a -> Exp b)        -- Scalar computation in Acc world
      -> Acc (Array sh a)        -- Array in Acc world
      -> Acc (Array sh b)        -- Array in Acc world
use :: Arrays arrays => arrays -> Acc arrays -- Copy to Acc world
run :: Arrays a => Acc a -> a    -- Run Acc computation, copy back
```

```
*Main> let arr = fromList (Z:.3:.5) [1..] :: Matrix Int
*Main> run $ A.map (+1) (use arr)
Matrix (Z :. 3 :. 5)
  [  2,  3,  4,  5,  6,
     7,  8,  9, 10, 11,
    12, 13, 14, 15, 16]
```

Scalar Arrays

Accelerate operates mostly with arrays.

unit converts a conventional scalar (in Exp-land) into a 0-D array:

```
*Main> :t unit
unit :: Elt e => Exp e -> Acc (Scalar e)
*Main> run $ unit (3 + 1 :: Exp Int)
Scalar Z [4]
```

Indexing in Acc

indexArray works in normal-land; *!* works in Acc-land.

```
*Main> let arr = fromList (Z:.3:.5) [1..] :: Matrix Int
*Main> arr
Matrix (Z :. 3 :. 5)
  [  1,  2,  3,  4,  5,
     6,  7,  8,  9, 10,
    11, 12, 13, 14, 15]

*Main> :t (!)
(!) :: (Shape sh, Elt e) => Acc (Array sh e) -> Exp sh -> Exp e
*Main> :t index2
index2 :: Elt i => Exp i -> Exp i -> Exp ((Z :. i) :. i)

*Main> run $ unit (use arr ! index2 2 1)
Scalar Z [12]
```

Initializing Arrays in Acc-land

```
*Main> :t generate
generate
  :: (Shape sh, Elt a) =>
     Exp sh -> (Exp sh -> Exp a) -> Acc (Array sh a)
*Main> :t unlift
unlift :: Unlift c e => c (Plain e) -> e

*Main> run $ generate (index2 3 5)
*Main|  (\ix -> let Z:.y:.x = unlift ix :: Z :. Exp Int :. Exp Int
*Main|          in x + 10 * y)
Matrix (Z :. 3 :. 5)
  [  0,  1,  2,  3,  4,
    10, 11, 12, 13, 14,
    20, 21, 22, 23, 24]
```

```haskell
type Weight = Int32
type Graph = Array DIM2 Weight
step :: Acc (Scalar Int) -> Acc Graph -> Acc Graph
step k g = generate (shape g) sp where
   k' = the k    -- Pass index as scalar array: avoids kernel recompilation
   sp :: Exp DIM2 -> Exp Weight
   sp ix = let (Z :. i :. j) = unlift ix -- Get index
           in A.min (g ! (index2 i j))
                    (g ! (index2 i k') + g ! (index2 k' j))

shortestPathsAcc :: Int -> Acc Graph -> Acc Graph
shortestPathsAcc n g0 = foldl1 (>->) steps g0  -- Pipeline operator
  where steps :: [ Acc Graph -> Acc Graph ]
        steps =  [ step (unit (constant k)) | k <- [0 .. n-1] ]

shortestPaths :: Graph -> Graph
shortestPaths g0 = run (shortestPathsAcc n (use g0))
  where Z :. _ :. n = arrayShape g0
```