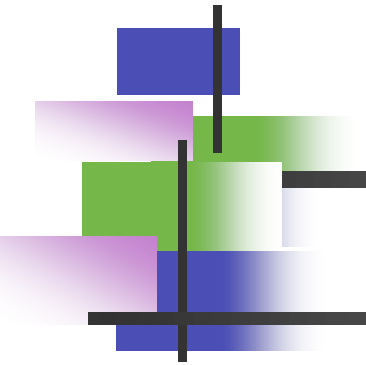
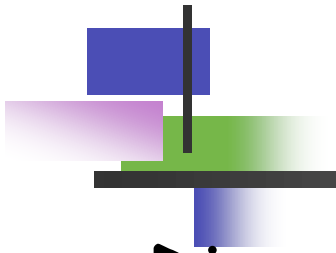


9. GPU Computing – Part 2

Parallel Programming

Dr Hamidreza Khaleghzadeh
School of Computing
University of Portsmouth



Goals

- Discuss various details of *CUDA* (and *OpenCL*) programming models, imposed by GPU architectural constraints.
- Java on GPUs – an example.
- Brief look at Deep Learning on GPUs.

THE DETAILS: THREAD BLOCKS AND ALL THAT

The Story So Far

- Aside from some messy details, like the need for the programmer to orchestrate transfer of data between CPU and GPU, the programming model introduced last lecture was fairly simple.
- In essence one defines a kernel function that runs in a thread on the GPU, then forks off many identical copies of this thread in a single call from the host.
- A thread can read or write different words in memory (array elements) based on its thread id.

Thread Blocks

- We have ignored one significant complication – CUDA threads are divided into *blocks*.
- A *thread block* is simply a collection of threads all running the same kernel.
- In the simple matrix multiplication earlier we put all threads in a *single* block.
  - This was specified by the line:

```
dim3 dimGrid(1, 1) ;
```

Previous Matrix Multiplication

```
void matmul(float* A, float* B, float* C, int N) {
    ... create arrays in device memory (see following slides) ...
    ... copy host arrays A and B to device arrays Ad and Bd ...

    // Set up the execution configuration
    dim3 dimBlock(N, N) ;
    dim3 dimGrid(1, 1) ;

    // Launch the device computation threads!
    matmulKernel<<<dimGrid, dimBlock>>>(Ad, Bd, Cd, N) ;

    ... copy device array Cd to host array C (see later) ...
}
```

Limitations on Thread Numbers

- While the CUDA programming model encourages use of *many* threads, any given GPU puts some limit on the *maximum number of threads in any single block*.
- If for example this limit is 1024, then the code provided earlier only works if $N \leq 32$ (because $32^2 = 1024$).

Large Matrices

- In our matrix multiplication example, we can easily use multiple blocks of threads, something like this in the host program:

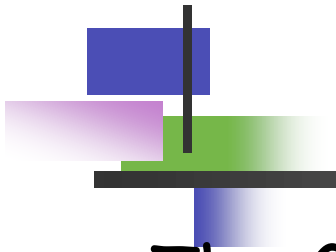
```
// Set up the execution configuration
dim3 dimBlock(32, 32) ;
dim3 dimGrid(N / 32, N / 32) ;
```

- assuming for simplicity *N* is a multiple of 32 - and this in the kernel function:

```
int tx = 32 * blockIdx.x + threadIdx.x ;
int ty = 32 * blockIdx.y + threadIdx.y ;
```

Generalizations

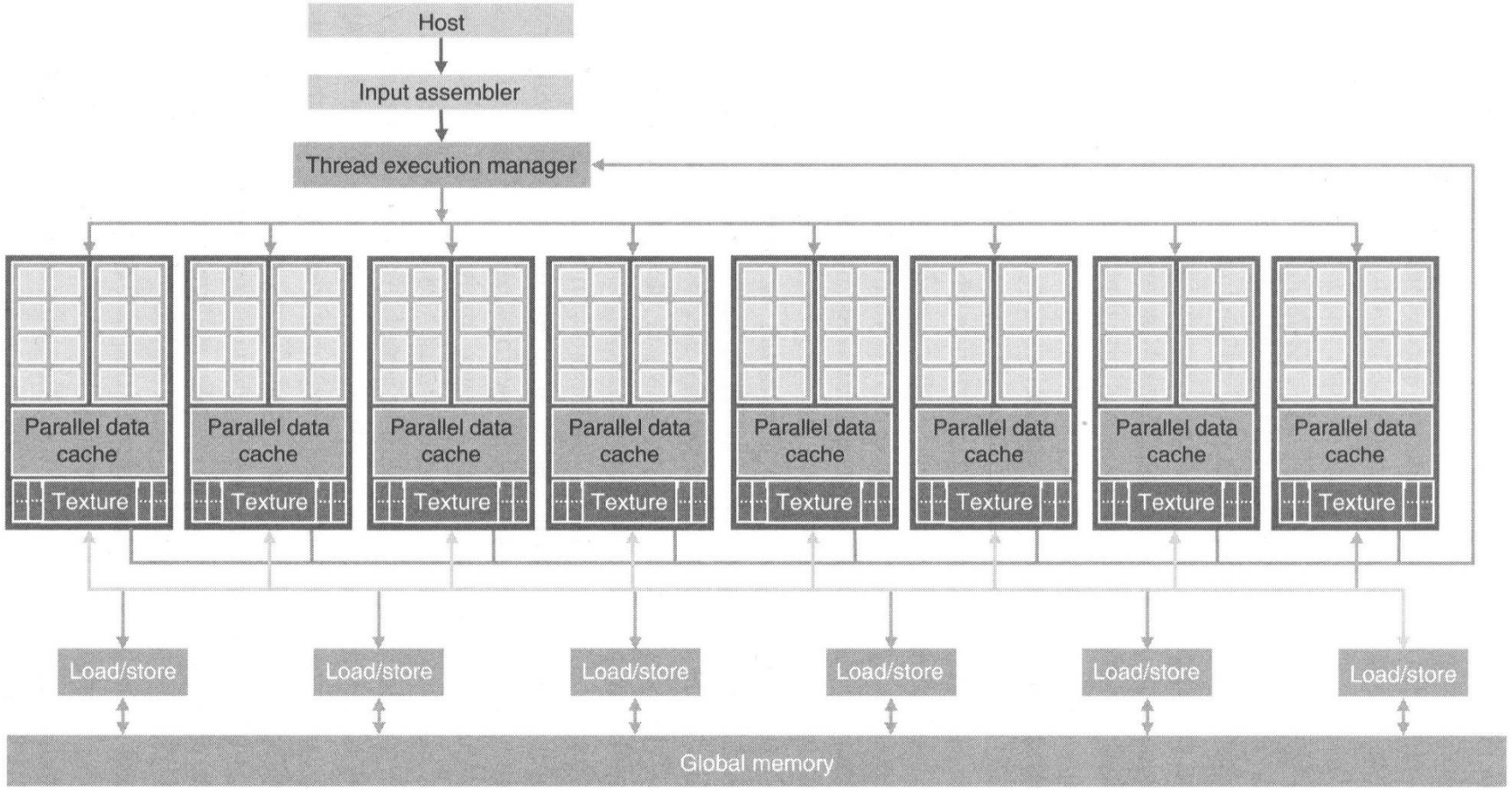
- There is no limit on the number of thread blocks so this will work fine. With very little extra work it can be generalized to deal with the limitation that *N* is a multiple of 32.
- But for some programs the limit on the number of threads in a block is a more serious obstacle.



Streaming Multiprocessors

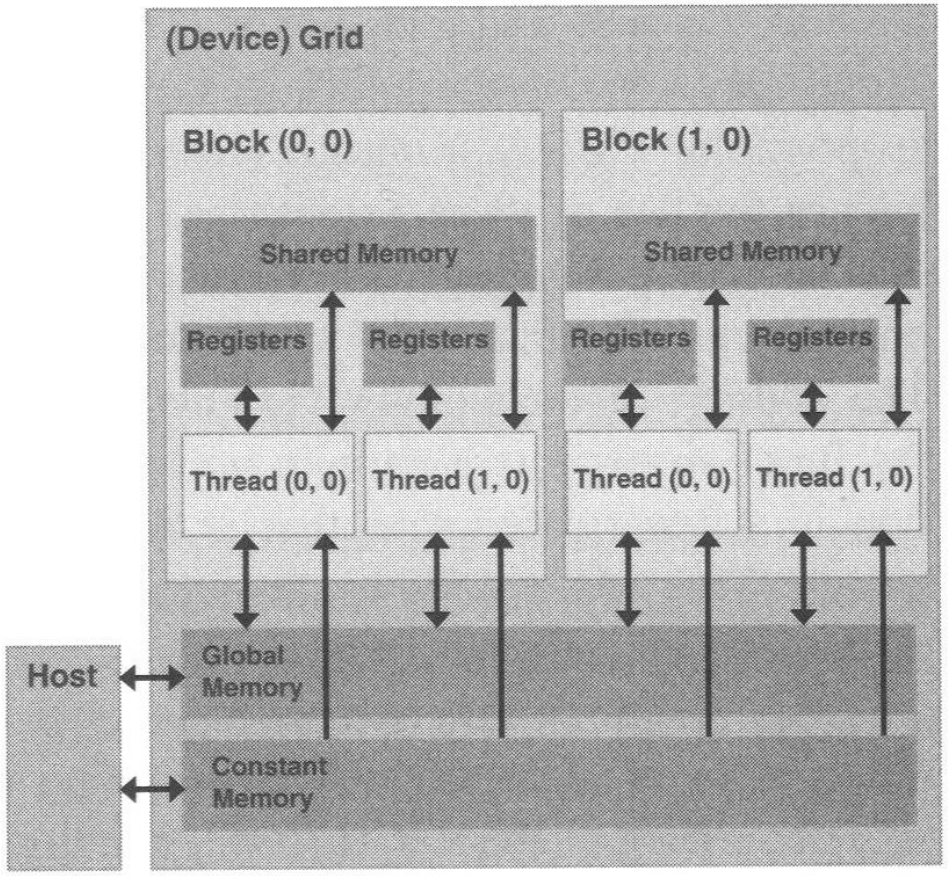
- The GPU consists of a number of *streaming multiprocessors* (SMs), each of which is an *SIMD parallel processor* in its own right (consisting of a number of processing elements).
- Each block is assigned to a particular SM.
- There is *fast shared memory* associated with each SM, and threads *within a block* can interact using this memory.
- Threads in a block can also perform barrier synchronizations amongst themselves.

Example GPU Reminder†



†Kirk & Hwu, PMPP, Fig 1.3

Blocks and Shared Memory†

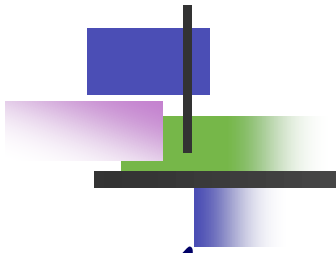


†Kirk & Hwu, PMPP, Fig 3.7

Limitations on Interaction of Threads

- But threads in *different* blocks potentially execute on *different* streaming multiprocessors – threads in different blocks *cannot* interact in a meaningful way.
- This apparently means something like our parallel Game of Life could not be written in anything like the way we wrote it before (with global barriers), if it were to span multiple blocks to make use of all the SMs on the GPU…

JAVA ON GPUS: A FIRST LOOK AT APARAPI



Aparapi

- *Aparapi* (*A PARallel API*) is one of the better known attempts at providing a Java API for running parallel code on GPUs.
  - Others include *JOCL*, *Rootbeer*, etc…
- Originally developed by AMD, now open source:
  - https://github.com/aparapi/aparapi

Example from Developers

```
final float inA[] = ....   // initialize somehow
final float inB[] = ....   //    “   “   (inA.length=inB.length)
final float result[] = new float[inA.length];
Kernel kernel = new Kernel(){
    public void run(){
        int i = getGlobalId();
        result[i]=inA[i]+inB[i];
    }
};
Range range = Range.create(result.length);
kernel.execute(range);
```

Comments

- Kernel code is run() method of class extending Kernel (analogy with standard Thread class).
  - Can expect Java code in run() method to be somewhat restricted.
- Typically define Kernel subclass as an anonymous nested class.
  - Can access local variables (arrays) of enclosing method for inputs and outputs.
- Range object replaces dimBlock, dimGrid of CUDA.
- execute() method of kernel runs "threads" (kernels) on GPU, and waits for completion.
  - Data transfer between CPU and GPU automatic.

An Experiment

- You need at least a machine with a GPU and device drivers for *OpenCL* that runs on the GPU
  - Aparapi programs will run on a Java Thread Pool otherwise – expect poor performance.

Aparapi version of GPU matmul

```
Kernel kernel = new Kernel() {
    public void run() {
        int tx = getGlobalId(0) ;
        int ty = getGlobalId(1) ;
        float sum = 0 ;
        for(int k = 0 ; k < N ; k++) {
            sum += a [N * ty + k] * b [N * k + tx] ;
        }
        c [N * ty + tx] = sum ;
    }
} ;
Range range = Range.create2D(N, N) ;
kernel.execute(range) ;
```

Comments

- Kernel very similar to original CUDA, but calling code *much* simpler.
- Arrays a, b, c automatically imported from enclosing scope – declared and initialized or output in main method.
- Supports multidimensional ranges – i.e. grids of threads
  - blocks dealt with automatically, it seems.
- Full code of this example will be on Moodle.

Compiling and Running the Code

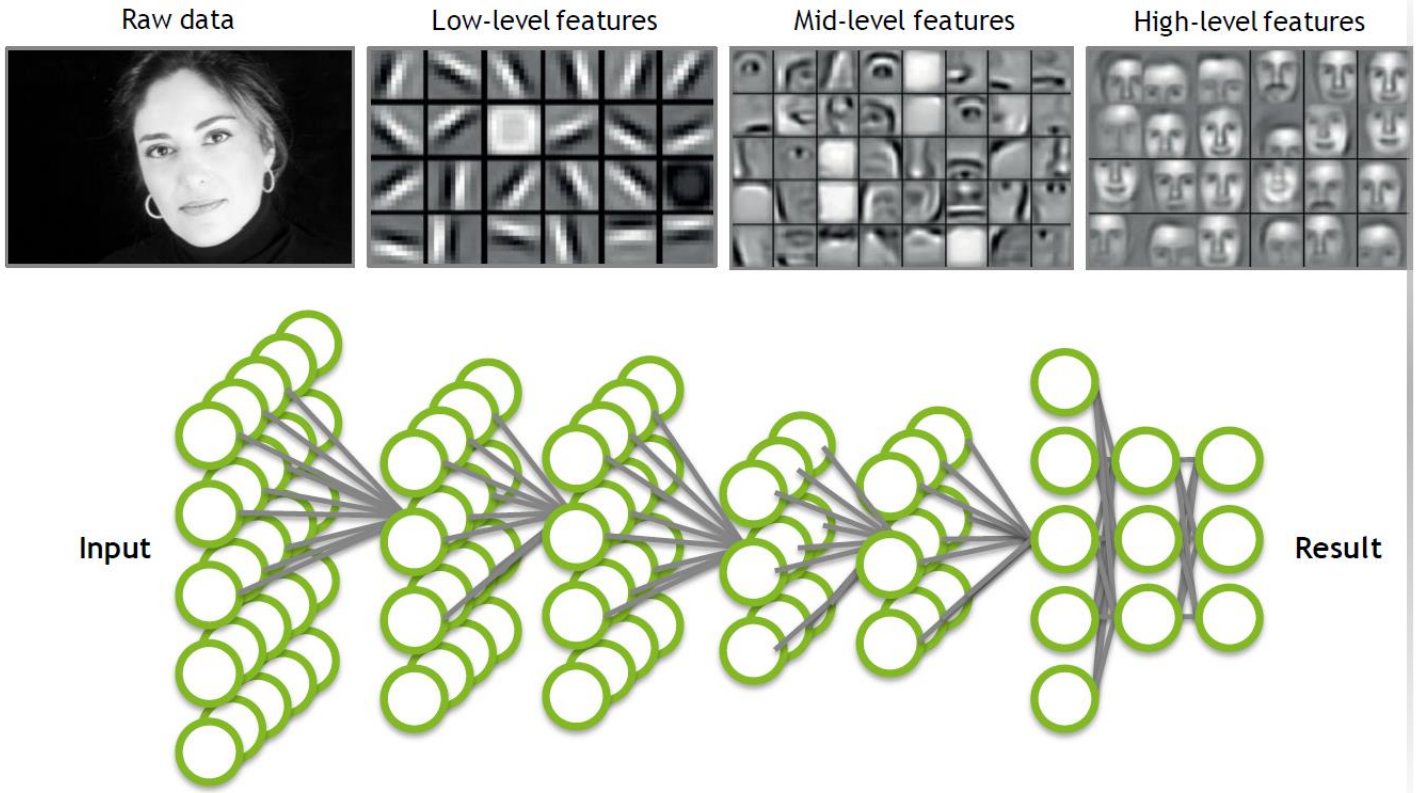
- In Netbeans, select Maven tab then Java Application. Add the Maven dependency given on Aparapi Web page.
- Didn't have to do anything else – it "just worked"!

GPUS FOR DEEP LEARNING

Deep Learning

- Last decade (often cited as starting around 2006) has seen rapid growth in development and deployment of *Deep Learning*.
- From outsider's perspective, looks much like a reimagining of classical *Multi-Layer Perceptrons* (artificial neural networks with multiple hidden layers), facilitated by recent technical "breakthroughs".
- One of these seems to be *harnessing of GPUs* to make training large networks practical.

Deep Neural Net (DNN)†



Application components:

Task objective
- e.g. Identify face

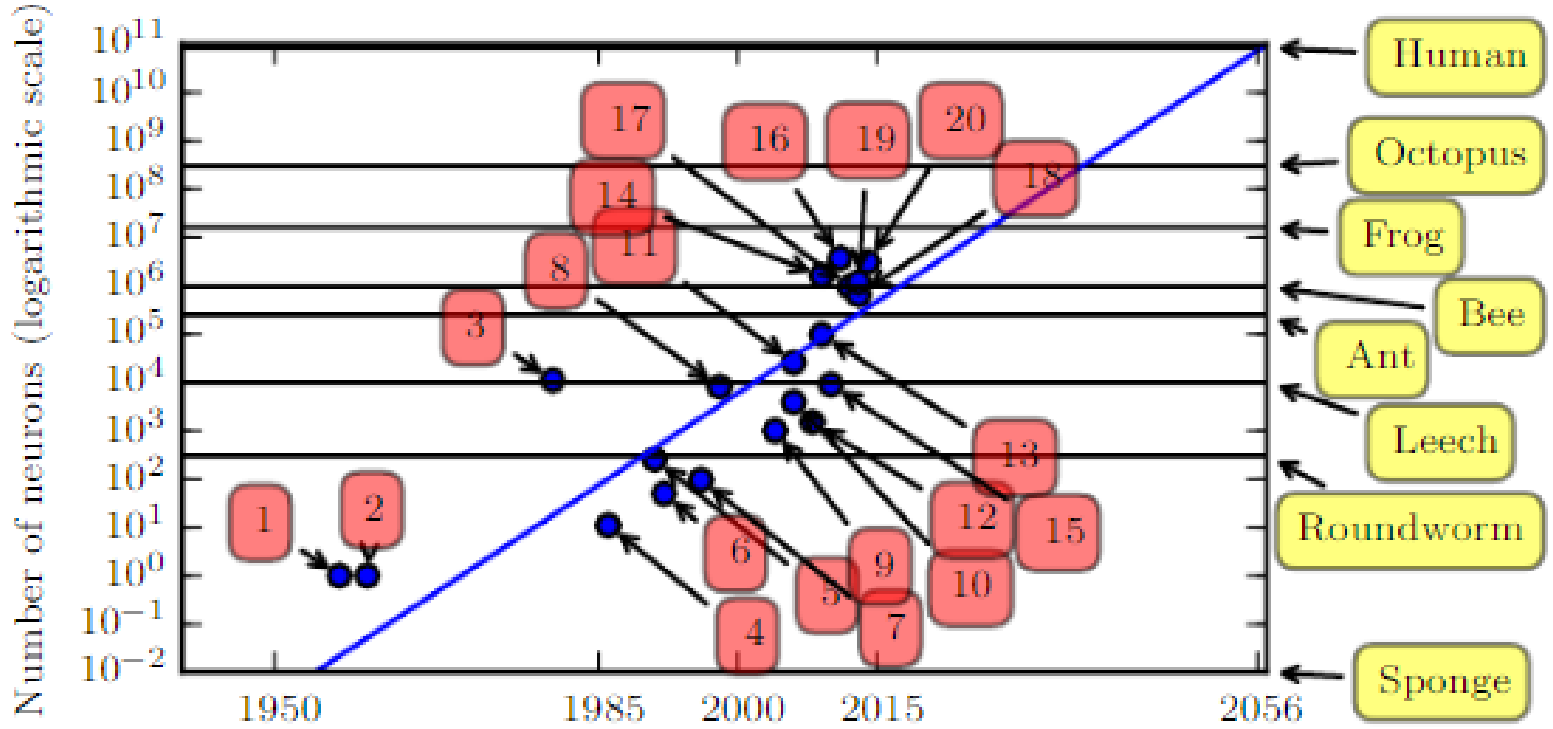
Training data
- 10-100M images

Network architecture
- ~10 layers
- 1B parameters

Learning algorithm
- ~30 Exaflops
- ~30 GPU days

†Source: *Deep Learning on GPUs*, NVIDIA, March 2016

Growth of Artificial Neural Networks†

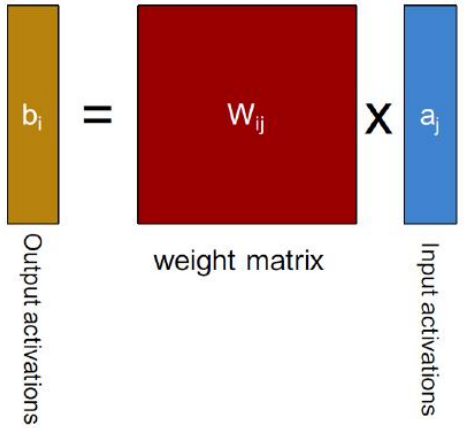


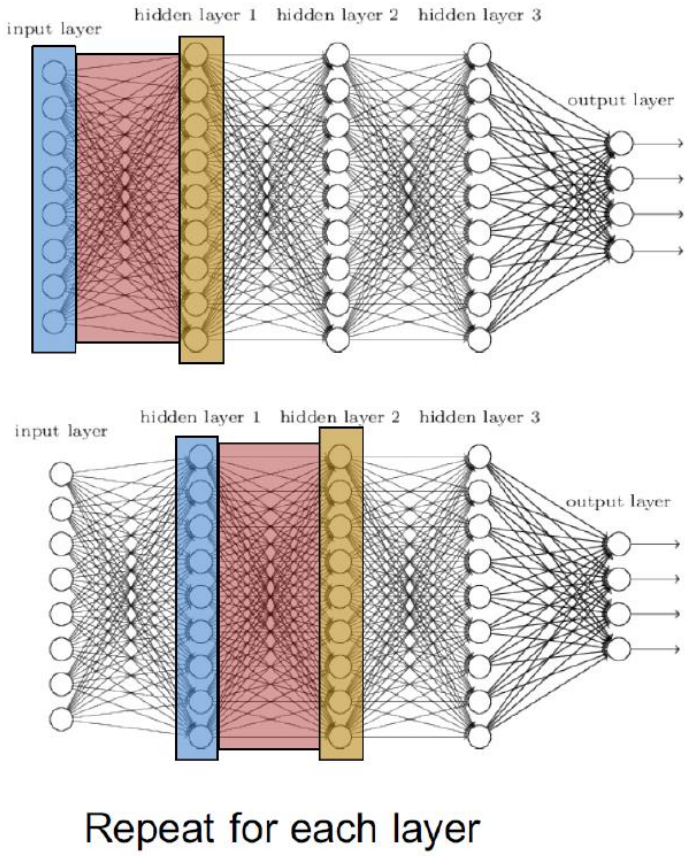
- Claimed to be "doubling every 2.4 years". See citation below, Figure 1.11, for details.

†Source: Goodfellow, Bengio & Courville, *Deep Learning*, MIT press, 2016

Classification using DNN†

Key operation is dense M x V



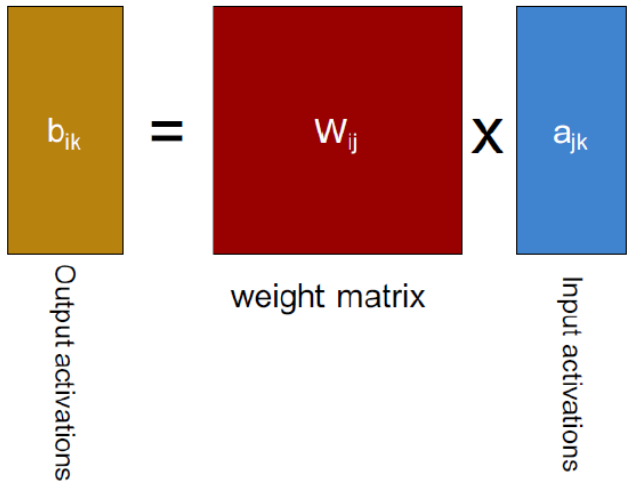


Backpropagation uses dense matrix-matrix multiply starting from softmax scores

†Source: *Deep Learning on GPUs*, NVIDIA, March 2016.

Batching Activation Processing†

Batching for training and latency insensitive.
M x M



Batched operation is M x M - gives re-use of weights.

Without batching, would use each element of Weight matrix once.

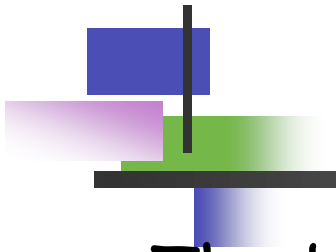
Want 10-50 arithmetic operations per memory fetch for modern compute architectures.

- Natural fit for training by Stochastic Gradient Descent.

†ibid.

GPU Enabled DL Environments

- *TensorFlow*, developed by Google.
- *PyTorch* – a Python-based descendant of earlier Torch library.
- *Caffe*, developed by Berkeley Vision and Learning Centre.
- NVIDIA *CuDNN* - GPU-accelerated library of primitives for deep neural networks in CUDA.
- *Deeplearning4j* – training deep neural networks using distributed GPUs under *Apache Spark*.
- Many others…



Summary

- The basic programming model of *CUDA* is quite straightforward, and ingenious as far as it goes.
- But in detail GPU programming is still quite hard to master due to specialized nature of hardware.
- *Aparapi* provides a significantly simplified programming model, but performance needs further investigation (projects!)
- One of the most promising application areas of GPUs today is in *Deep Learning*.

Further Reading

- David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 2nd edition, Morgan Kaufman, 2013 (*PMPP*).