Hardware and Software for NLP

Jeremy Appleyard, NVIDIA

Performance

Motivation

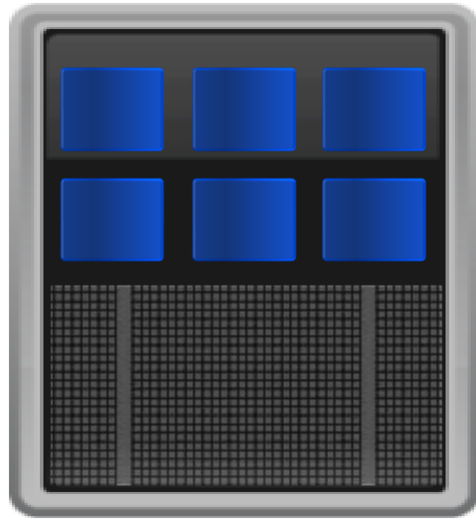
Decreasing time to solution is very useful:

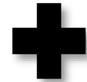
- For training it allows you to experiment with more model architectures
- In production environments providing a quick result to the end-user is crucial

Not everything can be done automatically: some things are up to the user

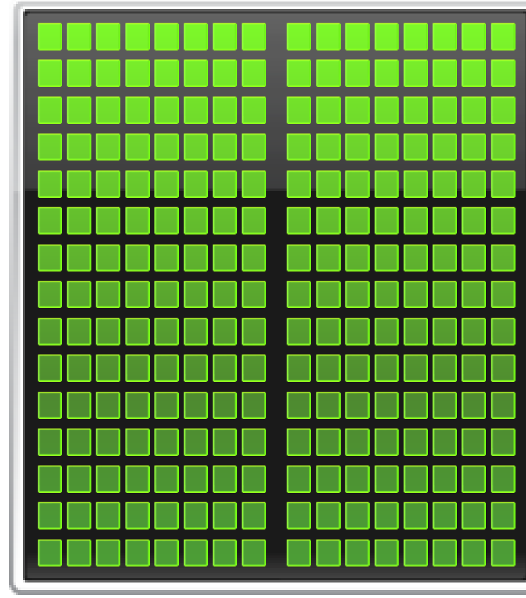
Accelerated Computing

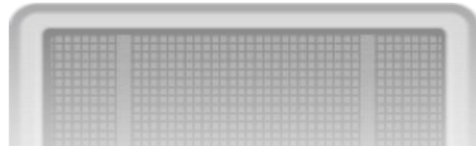
CPU
Optimized for
Serial Tasks



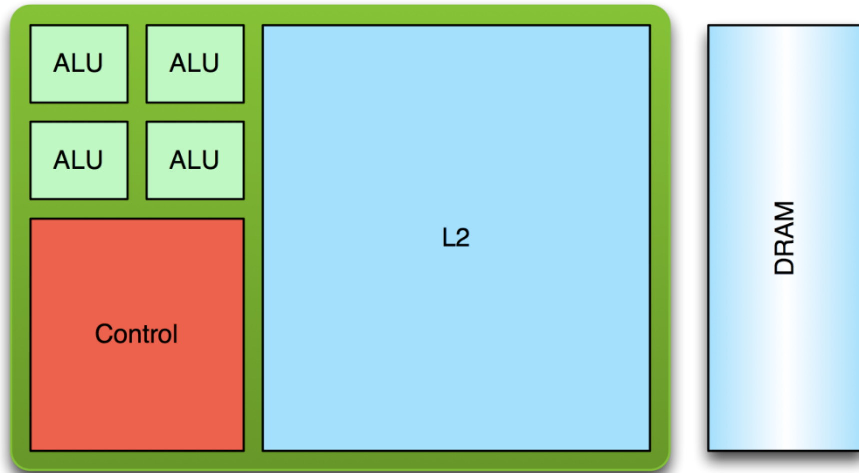


GPU Accelerator
Optimized for
Parallel Tasks



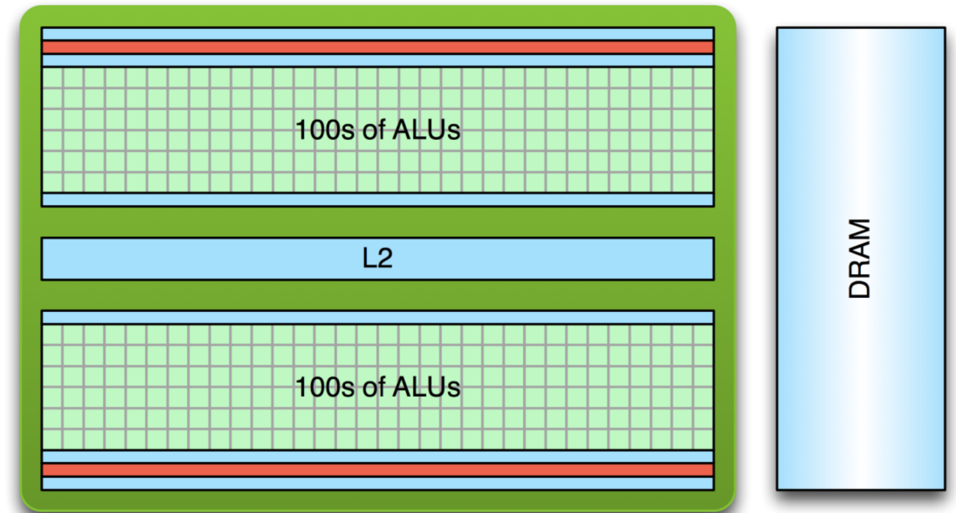


Low Latency or High Throughput?



CPU

- Optimized for low-latency access to cached data sets
- Control logic for out-of-order and speculative execution



GPU

- Optimized for data-parallel, throughput computation
- Architecture tolerant of memory latency
- More transistors dedicated to computation

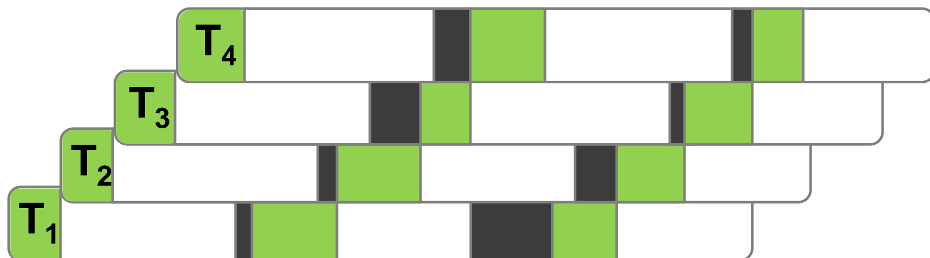
Low Latency or High Throughput

Design leads to performance

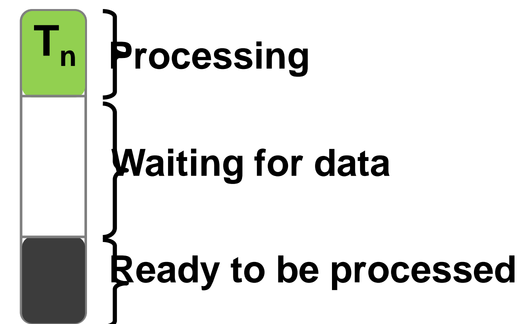
CPU architecture must minimize latency within each thread

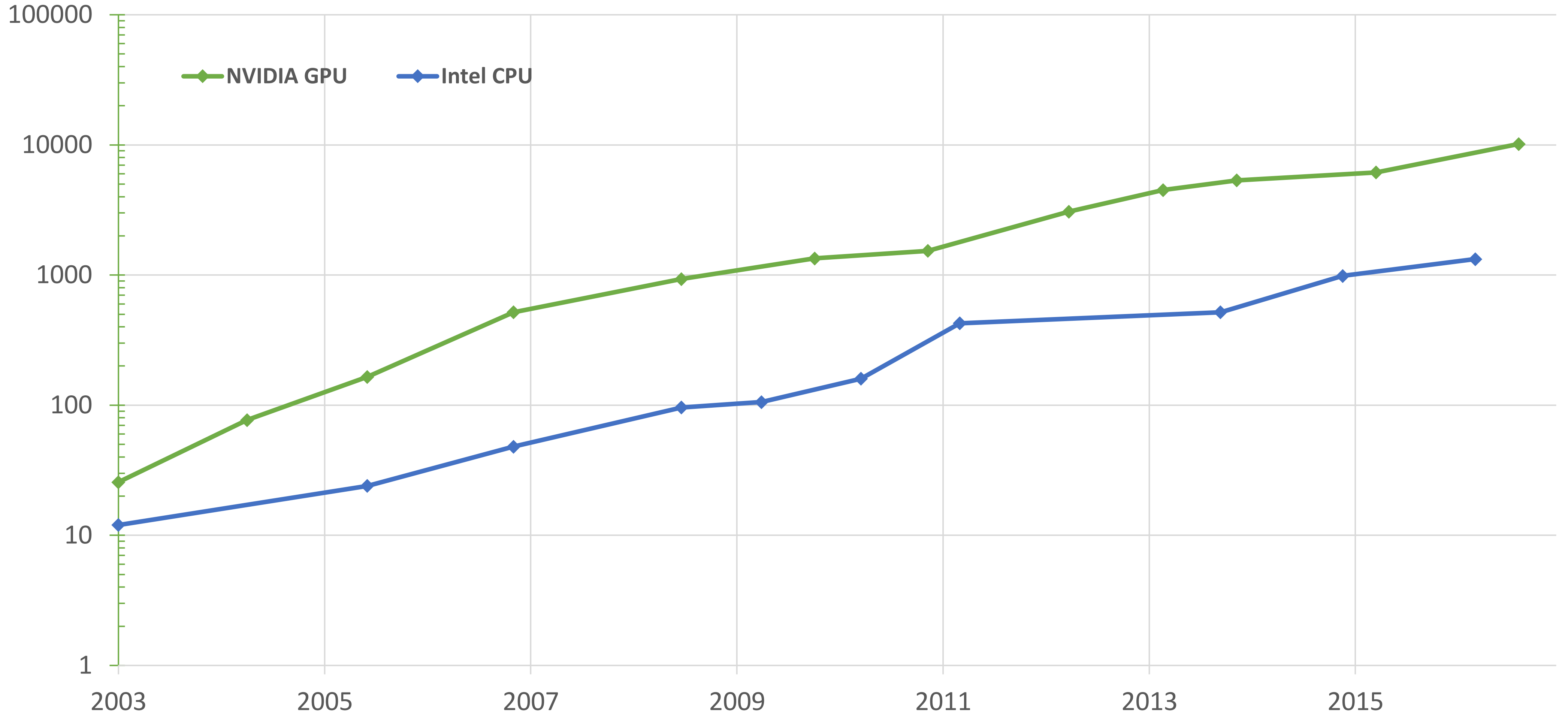
GPU architecture hides latency with computation (10+k threads)

GPU – High Throughput Processor

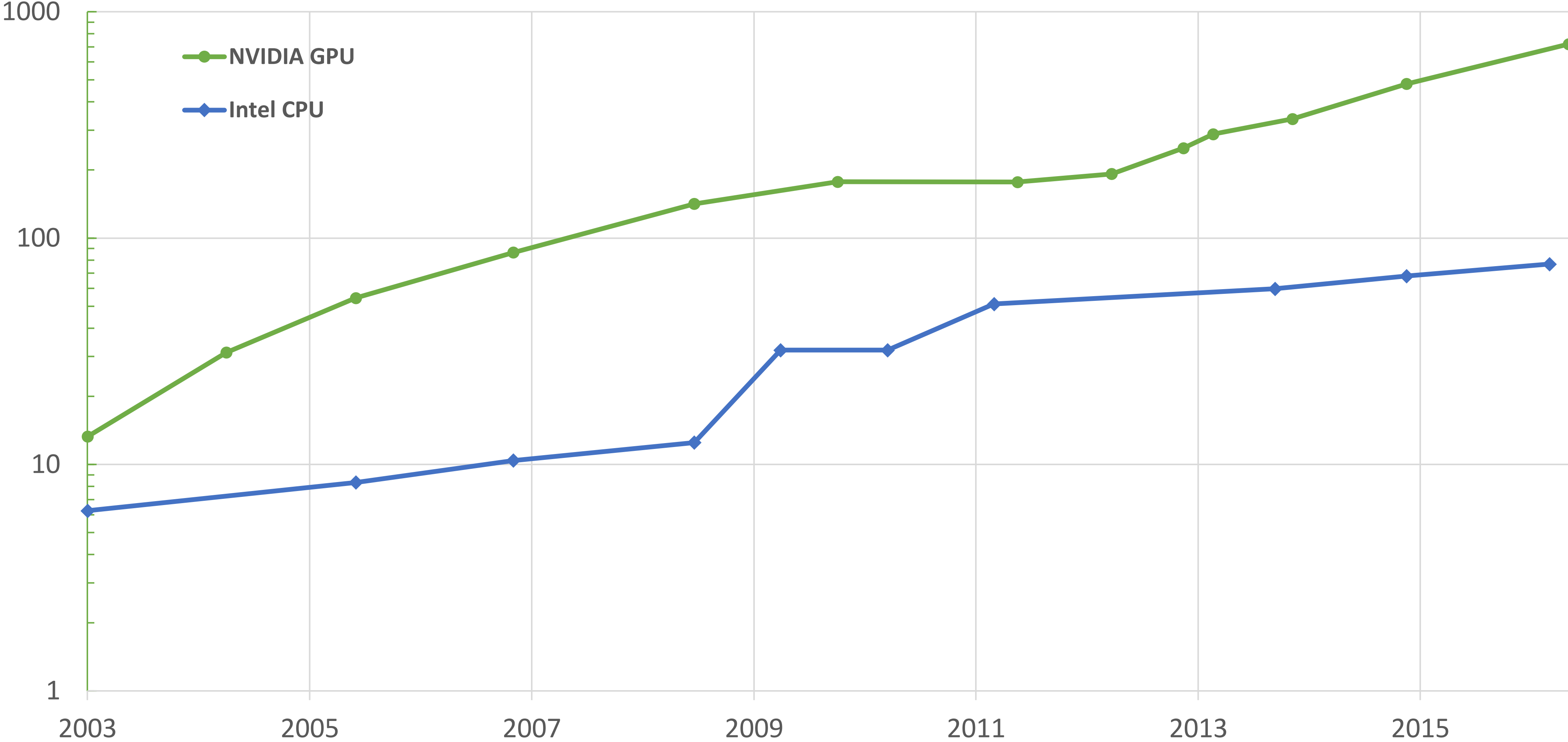


CPU core – Low Latency Processor





Theoretical single precision GFLOP/s at base clock
NVIDIA GPU
Intel CPU
100000
10000
1000
100
10
1
2003
2005
2007
2009
2011
2013
2015



Theoretical Peak GB/s
NVIDIA GPU
Intel CPU
1000
100
10
1
2003
2005
2007
2009
2011
2013
2015

Roofline Model

Useful analysis tool

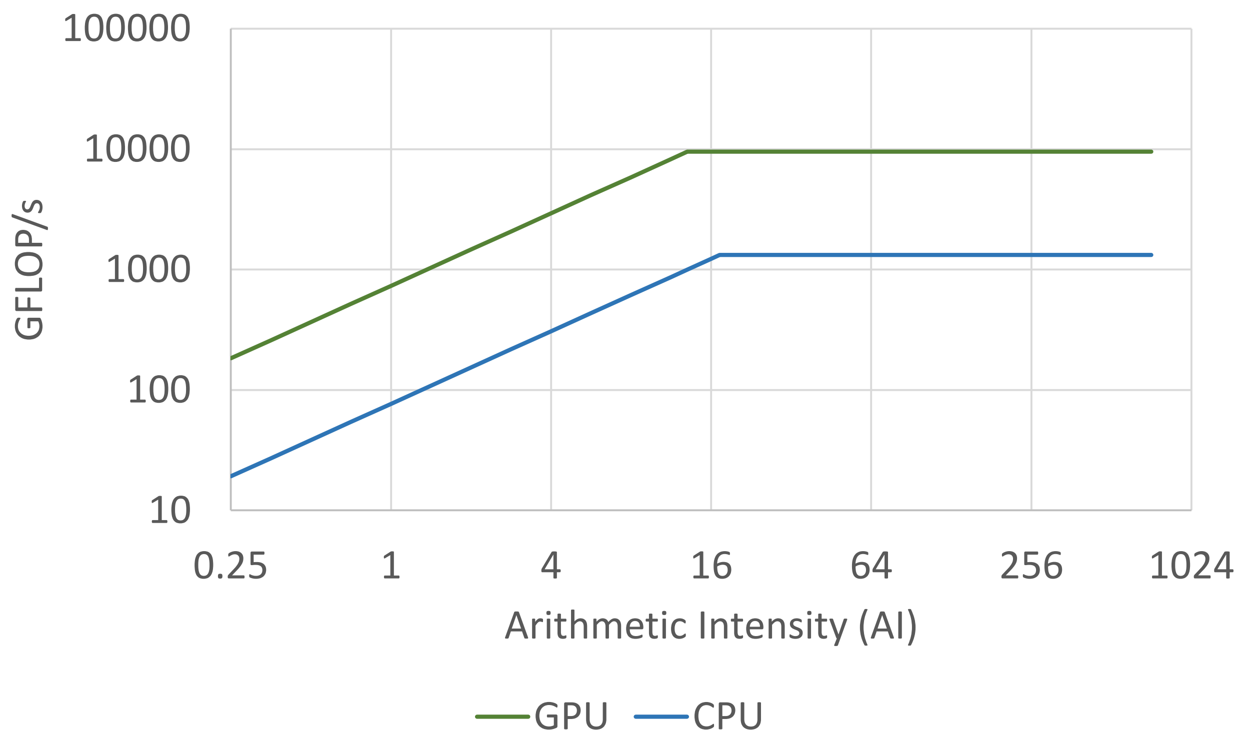
Floating point throughput is often the quantity we want to maximize. If we double achieved floating point throughput we double the amount of useful work we do in a given time.

A roofline model is a plot of the computational intensity of an algorithm against it’s expected floating point throughput for a given piece of hardware

Computational intensity is defined as the number of floating point operations per byte: Flops/Byte

Roofline Model

Current hardware



OPERATION	BYTES	FLOPS	AI
c=a+b	12	1	0.083
Mat-Vec	$O(N^2)$	$O(N^2)$	$O(1)$
1D FFT	$O(N)$	$O(N\log N)$	$O(\log N)$
Mat-Mat	$O(N^2)$	$O(N^3)$	$O(N)$
RNN	?	?	?

Performance Analysis of LSTM

LSTM

Computation required

Equations:

$$i_t = \sigma(W_i[w_t; h_{t-1}] + b_i)$$
$$f_t = \sigma(W_f[w_t; h_{t-1}] + b_f)$$
$$o_t = \sigma(W_o[w_t; h_{t-1}] + b_o)$$
$$\hat{c}_t = \tanh(W_c[w_t; h_{t-1}] + b_c)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t$$
$$h_t = o_t \circ \tanh(W_h c_t + b_h)$$

For a batch of 1, w_t, h_t are vectors of size H

Operations:

4x Matrix-vector multiplications (input 2H, output H)

1x Matrix-vector multiplication (input H, output H)

2x Pointwise tanh (size H)

3x Pointwise sigmoid (size H)

6x Pointwise add (size H)

2x Pointwise multiplication (size H)

LSTM

Computation required

Equations:

$$i_t = \sigma(W_i[w_t; h_{t-1}] + b_i)$$

$$f_t = \sigma(W_f[w_t; h_{t-1}] + b_f)$$

$$o_t = \sigma(W_o[w_t; h_{t-1}] + b_o)$$

$$\hat{c}_t = \tanh(W_c[w_t; h_{t-1}] + b_c)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t$$

$$h_t = o_t \circ \tanh(c_t)$$

For a batch of 1, w_t, h_t are vectors of size H

Operations:

4x Matrix-vector multiplications (input 2H, output H)

~~1x Matrix-vector multiplication (input H, output H)~~

2x Pointwise tanh (size H)

3x Pointwise sigmoid (size H)

5x Pointwise add (size H)

2x Pointwise multiplication (size H)

NVIDIA.

LSTM

Computation required

Equations:

$$i_t = \sigma(W_i[w_t; h_{t-1}] + b_i)$$
$$f_t = \sigma(W_f[w_t; h_{t-1}] + b_f)$$
$$o_t = \sigma(W_o[w_t; h_{t-1}] + b_o)$$
$$\hat{c}_t = \tanh(W_c[w_t; h_{t-1}] + b_c)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t$$
$$h_t = o_t \circ \tanh(c_t)$$

For a batch of B, w_t, h_t are matrices of size HxB.

Operations:

4x Matrix-matrix multiplications (input 2HxB, output HxB)

~~1x Matrix-vector multiplication (input H, output H)~~

2x Pointwise tanh (size HxB)

3x Pointwise sigmoid (size HxB)

5x Pointwise add (size HxB)

2x Pointwise multiplication (size HxB)

Increasing Parallelism

An aside

$$[A_1][h] = [x_1]$$

$$[A_2][h] = [x_2]$$

$$[A_3][h] = [x_3]$$

$$[A_4][h] = [x_4]$$

$$\begin{bmatrix} \\ A \\ \\ \end{bmatrix} [h] = \begin{bmatrix} \\ x \\ \\ \end{bmatrix}$$

As the matrix multiplications share inputs we can combine them

NVIDIA.

LSTM

Computation required

Equations:

$$i_t = \sigma(W_i[w_t; h_{t-1}] + b_i)$$
$$f_t = \sigma(W_f[w_t; h_{t-1}] + b_f)$$
$$o_t = \sigma(W_o[w_t; h_{t-1}] + b_o)$$
$$\hat{c}_t = \tanh(W_c[w_t; h_{t-1}] + b_c)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t$$
$$h_t = o_t \circ \tanh(c_t)$$

For a batch of B, w_t, h_t are matrices of size HxB.

Operations:

1x Matrix-matrix multiplication (input 2HxB, output 4HxB)

~~1x Matrix-vector multiplication (input H, output H)~~

2x Pointwise tanh (size HxB)

3x Pointwise sigmoid (size HxB)

5x Pointwise add (size HxB)

2x Pointwise multiplication (size HxB)

NVIDIA.

LSTM

Computation required

Equations:

$$i_t = \sigma(W_i[w_t; h_{t-1}] + b_i)$$

$$f_t = \sigma(W_f[w_t; h_{t-1}] + b_f)$$

$$o_t = \sigma(W_o[w_t; h_{t-1}] + b_o)$$

$$\hat{c}_t = \tanh(W_c[w_t; h_{t-1}] + b_c)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t$$

$$h_t = o_t \circ \tanh(c_t)$$

For a batch of B, w_t, h_t are matrices of size HxB.

Operations:

1x Matrix-matrix multiplication (input 2HxB, output 4HxB)

2x Pointwise tanh (size HxB)

3x Pointwise sigmoid (size HxB)

5x Pointwise add (size HxB)

2x Pointwise multiplication (size HxB)

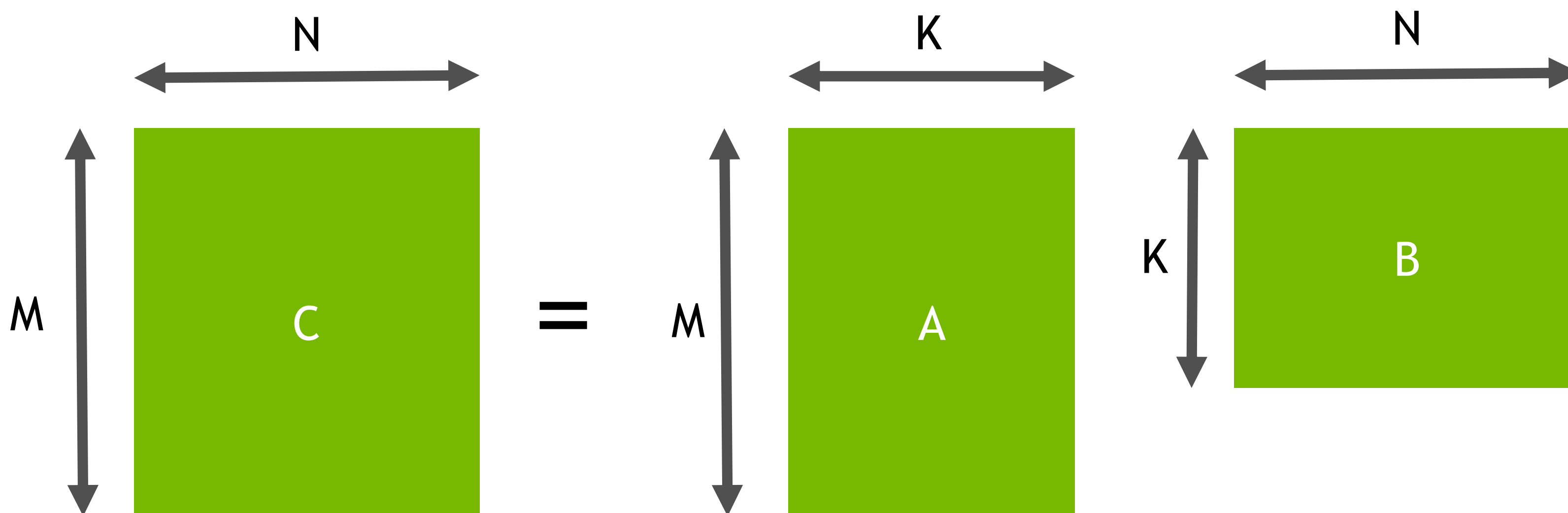
Matrix-matrix multiplications

Known as GEMM

A matrix-matrix multiplication (or GEMM) C=AB can be parameterized by three matrix dimensions: M, N and K.

Floating point ops = MN(2K-1) ≈ 2MNK

Bytes through memory = sizeof(dataType) * (MK + KN + MN)



LSTM Matrices

FLOP:Byte ratio

From before, data input is shape 2HxB, data output is shape 4HxB.

If A is our parameter matrix, this gives M=4H, N=B, K=2H

⇒ Floating point ops ≈ 2*4H*B*2H = 16HHB

⇒ Bytes through memory (fp32) = 4(8HH + 2HB + 4HB)

This gives a flops:byte ratio of 16HHB:4(8HH + 2HB + 4HB) = 2HB:3B+4H

NVIDIA.

Memory vs FLOP bound

Expected batch size

Batch size (B) can vary for a given model while giving a similarly performing model.

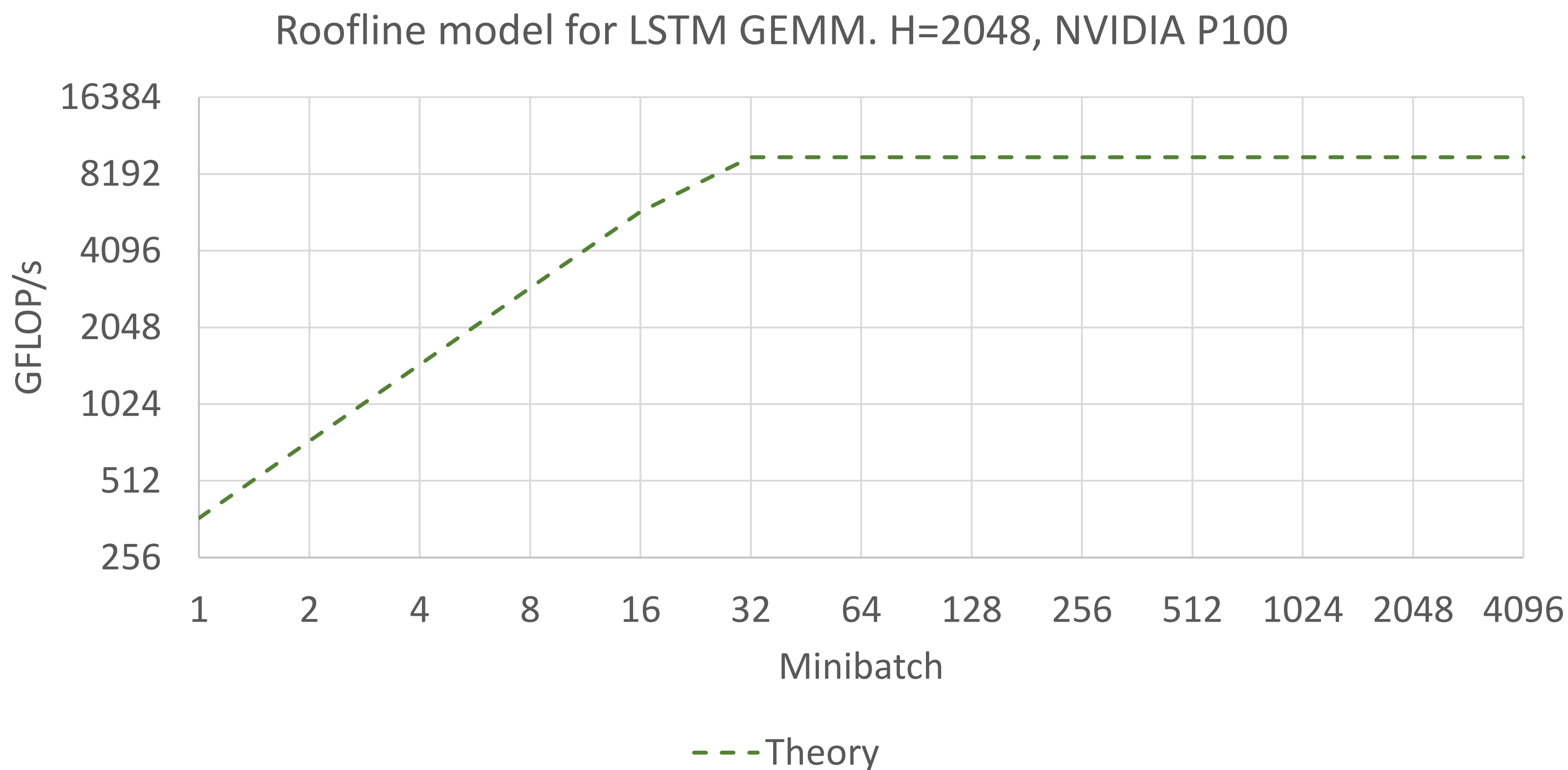
When training memory capacity required scales roughly linearly with batch size.

Convergence can be poor if batch size is too large or too small.

In deployment it may be only a few samples are available to be processed at once which potentially limits batching

Roofline model for LSTM

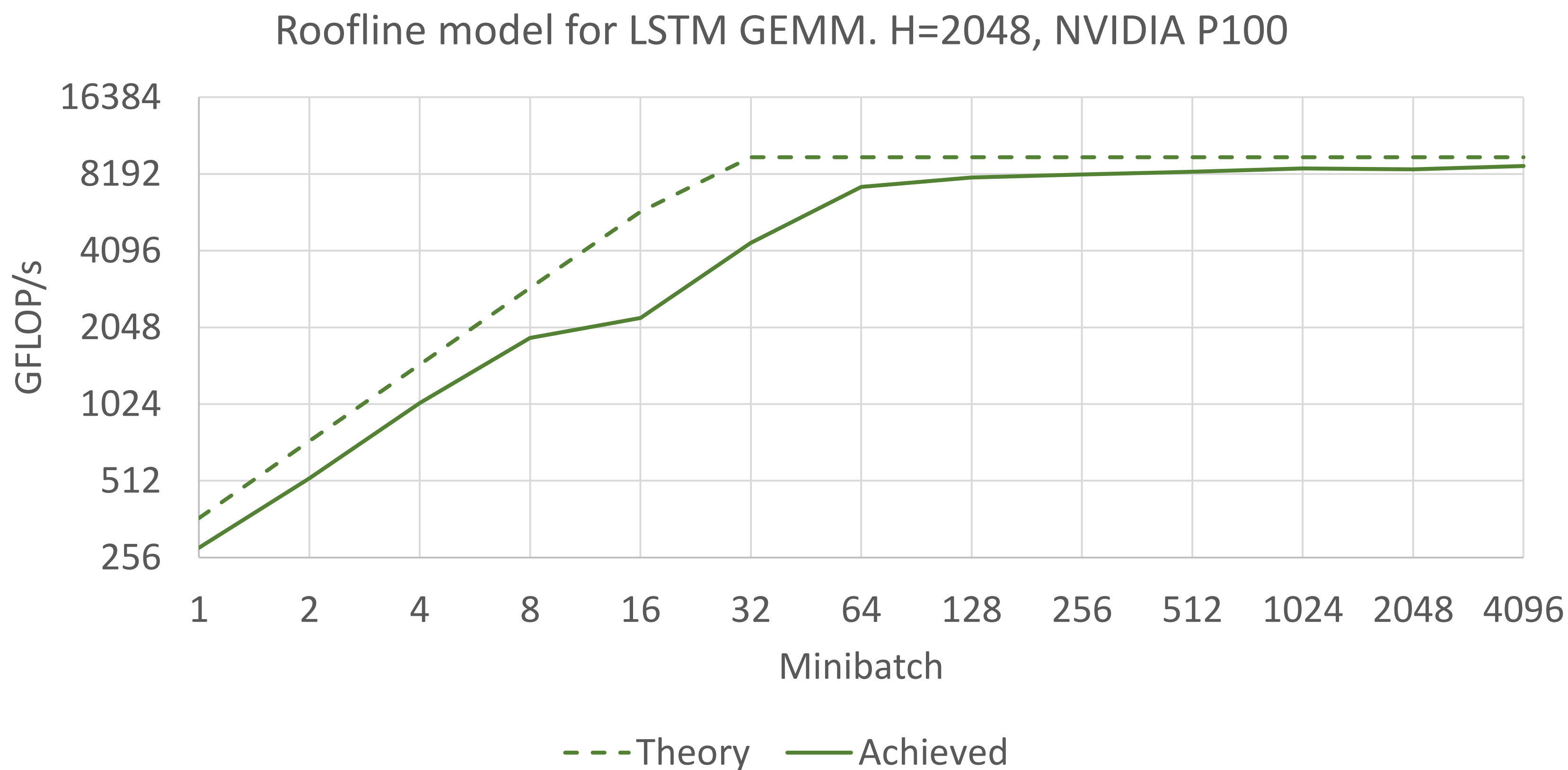
Selecting an efficient minibatch size



NVIDIA.

Roofline model for LSTM

Selecting an efficient minibatch size



NVIDIA.

LSTM Network Level Optimizations

Network Level Optimizations

Problem statement: given a fixed H and B, how can I make my network execute faster?

I will talk about three optimizations that could be performed:

1. Reducing memory traffic
2. Reducing overheads
3. Increasing parallelism

See [1] for other possible optimizations

[1]Optimizing Performance of Recurrent Neural Networks on GPUs. Appleyard et al., arXiv 2016.

NVIDIA.

Reducing Memory Traffic

Optimization #1

For small (unchangeable) minibatch we are bandwidth limited. The majority of the bandwidth is loading the A matrix, which is constant over time.

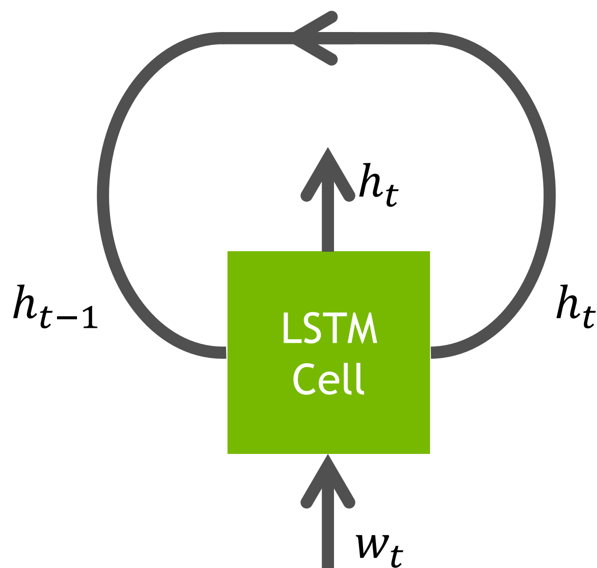
If we can reduce the amount of times we load A, the faster we expect to go.

NVIDIA.

Unrolling Over Time

Before

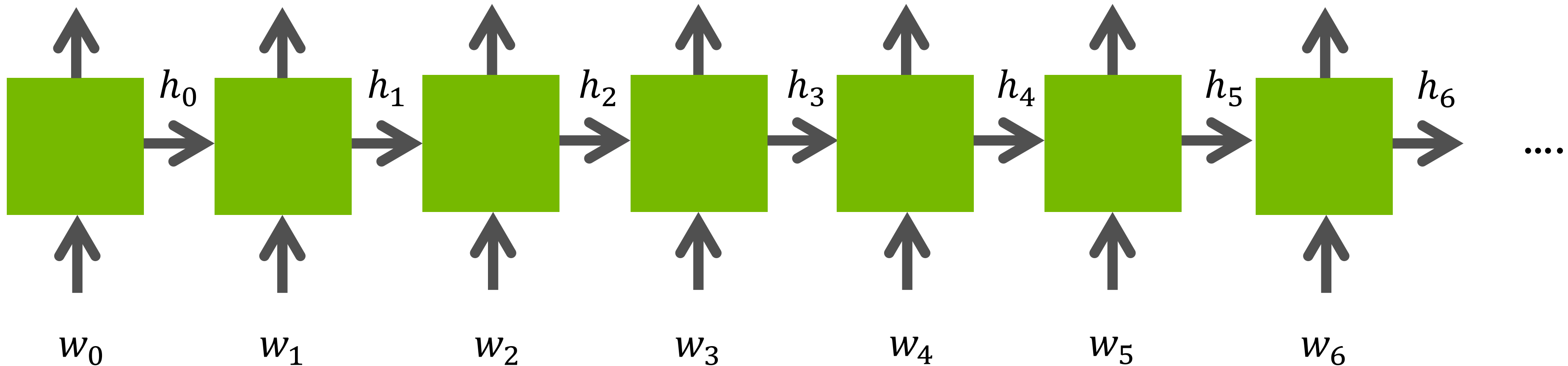
- Perform the same operation repeatedly on a combination of:
  - Previous state
  - New input



Unrolling Over Time

After

- Perform the same operation repeatedly on a combination of:
  - Previous state
  - New input



Freeing Dependencies

Previous Layer Input vs Recurrent Input

The LSTM equations have many matrix multiplications of the form: $W_*[w_t; h_{t-1}]$

Here w_t is the input from the previous layer, h_{t-1} the input from the previous recurrent step

We can isolate these two parts of the matrix multiplication:

$$W_*[w_t; h_{t-1}] = W_{l*}[w_t] + W_{r*}[h_{t-1}]$$

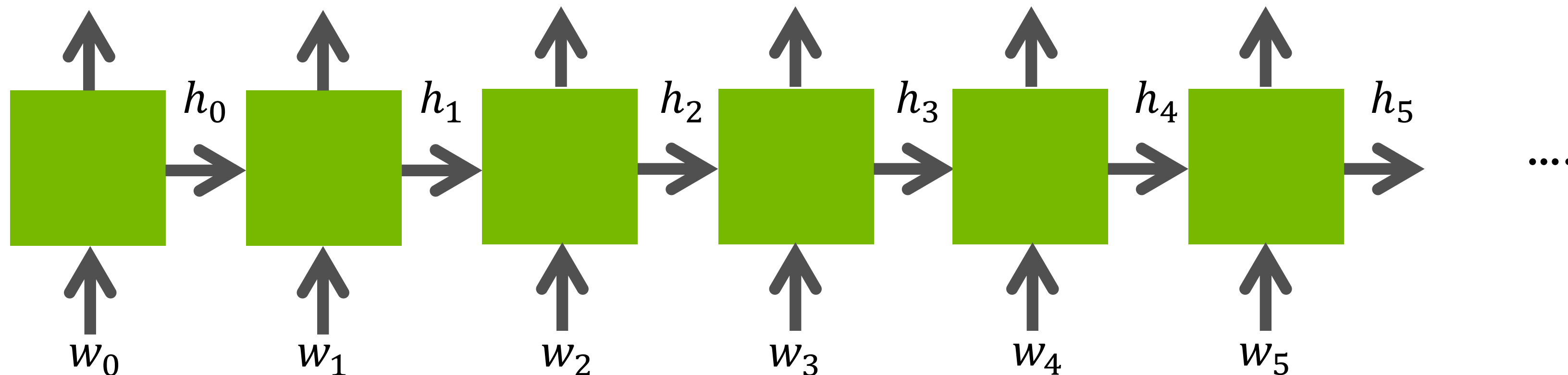
This results in one matrix multiplication operating on the output from the previous layer and one operating on the output from the previous step.

Functionally this is the same.

Fusing Operations Over Time

Effective Minibatch Increase

Each arrow can be seen as a matrix multiplication. Because the vertical arrows are independent, they can be grouped



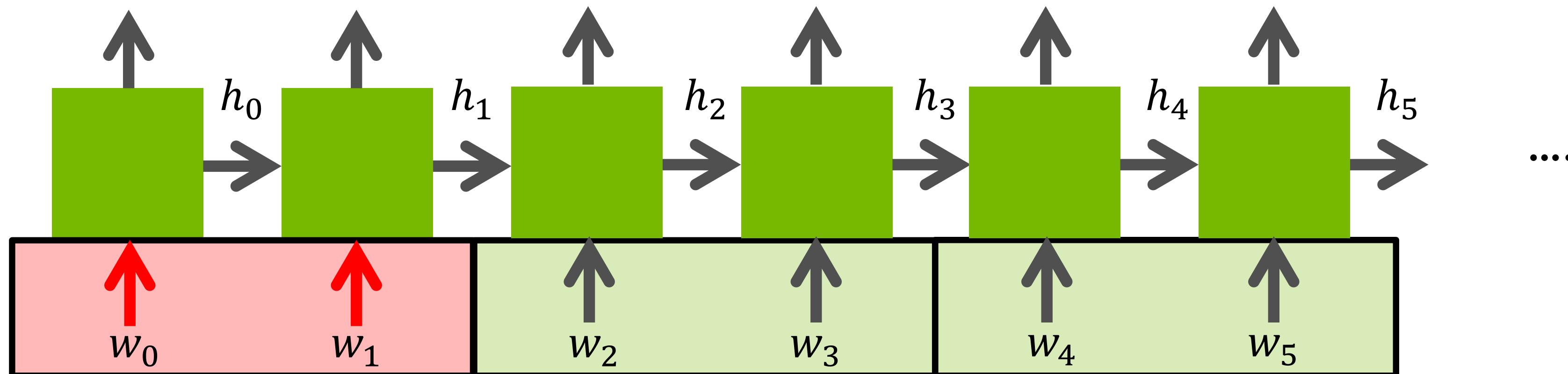
Fusing Operations Over Time

Effective Minibatch Increase

Each arrow can be seen as a matrix multiplication. Because the vertical arrows are independent, they can be grouped

With x times as many input elements to process, grouping by x is equivalent to increasing minibatch by x (parameter matrix is time-invarient).

Parallelism can be presever by task parallelism or "streaming"



NVIDIA.

Persistent RNNs

Advanced Technique

Fusing operations over time only helps some of our matrix multiplications: the recurrent operations are still bandwidth limited.

Diamos et al.[2] have proposed a method to keep the recurrent matrix in on-chip at a very high read bandwidth

Some constraints to implementation due to size limits of on-chip memory, but impressive speedups for small batches (>10x)

[2] Persistent RNNs: Stashing Recurrent Weights On-Chip, Diamos et al., ICML 2016

Overhead Reduction

Optimization #2

Equations:

$$i_t = \sigma(W_i[w_t; h_{t-1}] + b_i)$$

$$f_t = \sigma(W_f[w_t; h_{t-1}] + b_f)$$

$$o_t = \sigma(W_o[w_t; h_{t-1}] + b_o)$$

$$\hat{c}_t = \tanh(W_c[w_t; h_{t-1}] + b_c)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t$$

$$h_t = o_t \circ \tanh(c_t)$$

For a batch of B, w_t, h_t are matrices of size HxB.

Operations:

1x Matrix-matrix multiplication (input 2HxB, output 4HxB)

2x Pointwise tanh (size HxB)

3x Pointwise sigmoid (size HxB)

5x Pointwise add (size HxB)

2x Pointwise multiplication (size **HxB)**

NVIDIA.

Overhead Reduction

Optimization #2

Equations:

$$i_t = \sigma(W_i[w_t; h_{t-1}] + b_i)$$

$$f_t = \sigma(W_f[w_t; h_{t-1}] + b_f)$$

$$o_t = \sigma(W_o[w_t; h_{t-1}] + b_o)$$

$$\hat{c}_t = \tanh(W_c[w_t; h_{t-1}] + b_c)$$

$$c_t = f_t \circ c_{t-1} + i_t \circ \hat{c}_t$$

$$h_t = o_t \circ \tanh(c_t)$$

For a batch of B, w_t, h_t are matrices of size HxB.

Operations:

1x Matrix-matrix multiplication (input 2HxB, output 4HxB)

2x Pointwise tanh (input HxB)

2x Pointwise sigmoid (input HxB)

5x Pointwise add (input HxB)

2x Pointwise multiplication (input HxB)

Pointwise Operations

Optimization #2

A naïve implementation of these pointwise operations on the GPU would implement each as a separate GPU kernel

A kernel essentially means:

- CPU launches the kernel to the GPU with some small overhead
- For each pointwise element, launch one thread
- This thread reads the value it is responsible for, does a simple operation and writes its result

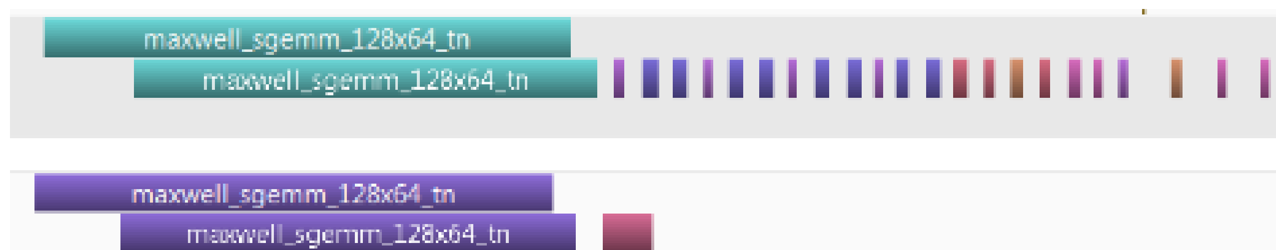
There are two problems with this: overhead of kernel setup and bandwidth

Pointwise Operations

Solution

A solution to this is to fuse the pointwise operations into one kernel

Instead of launching on kernel per operation, launch one to do the entire series

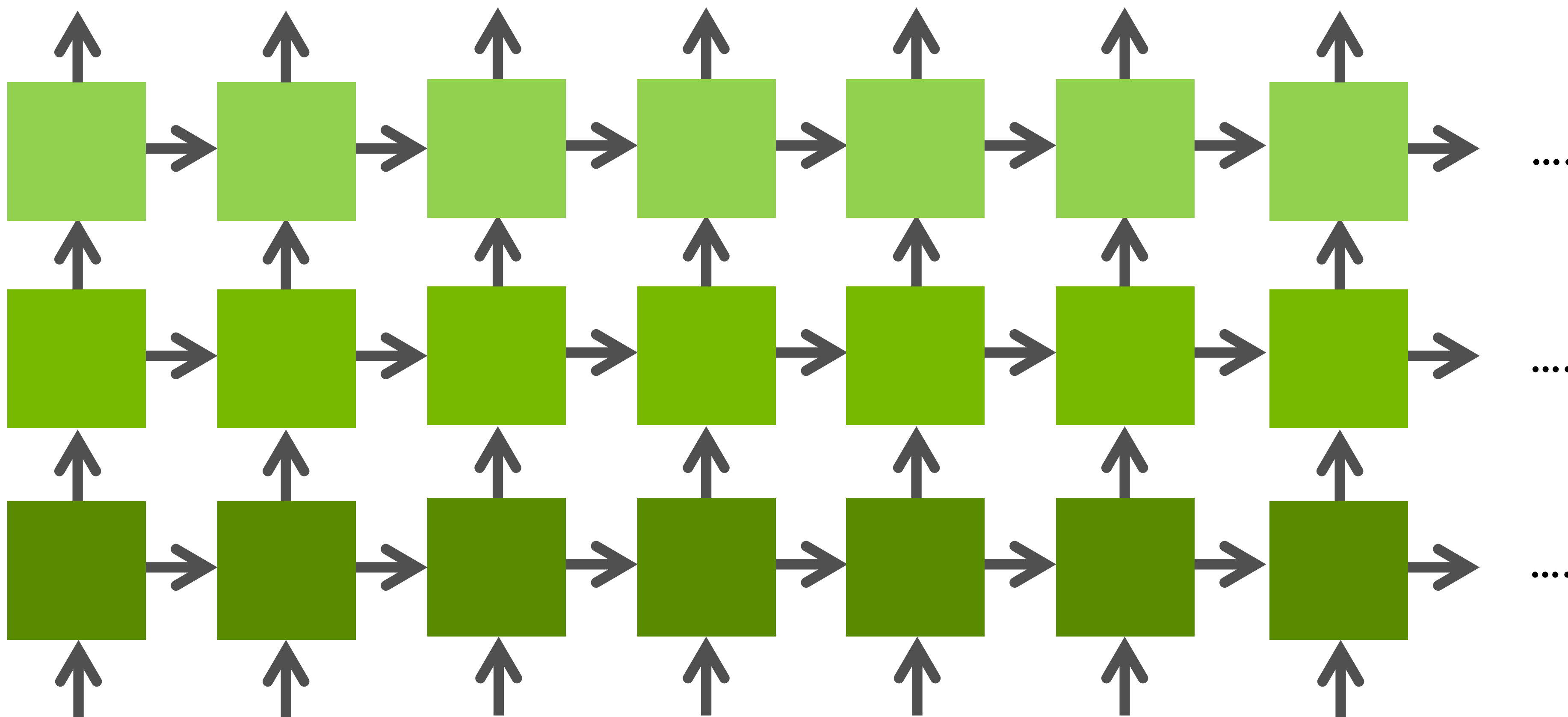


Source: https://devblogs.nvidia.com/parallelforall/optimizing-recurrent-neural-networks-cudnn-5/

In the above case, this is more than 2x speedup!

Increasing Parallelism

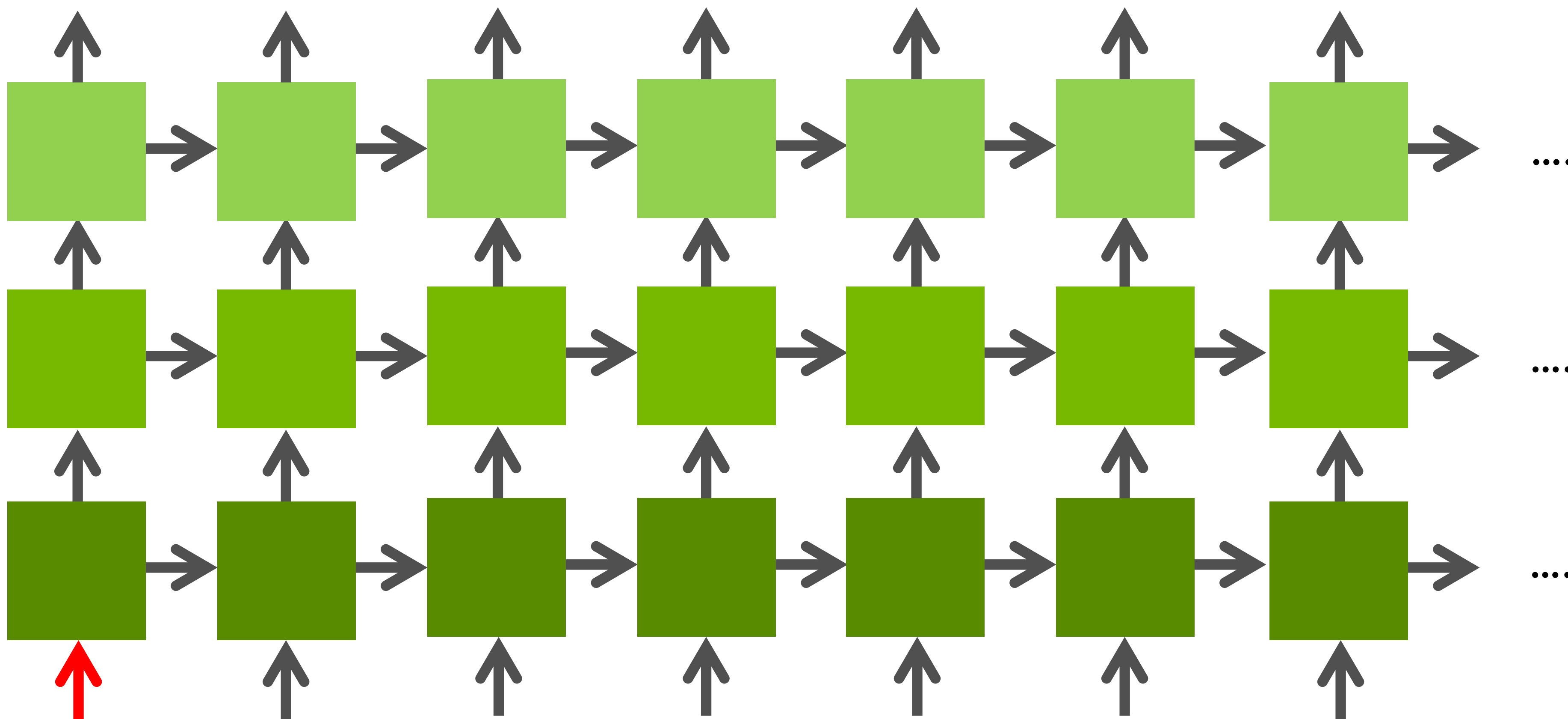
Optimization #3



NVIDIA.

Increasing Parallelism

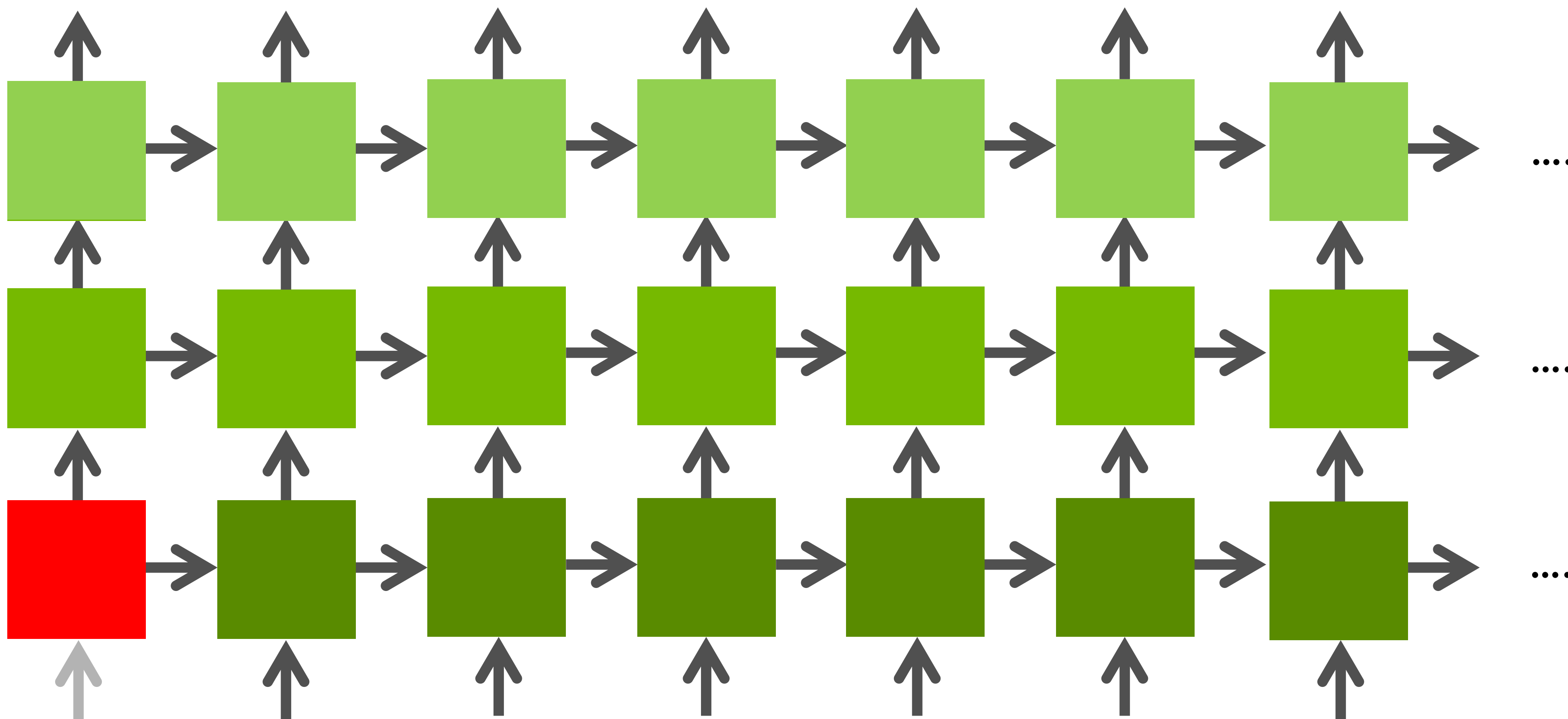
Optimization #3



NVIDIA.

Increasing Parallelism

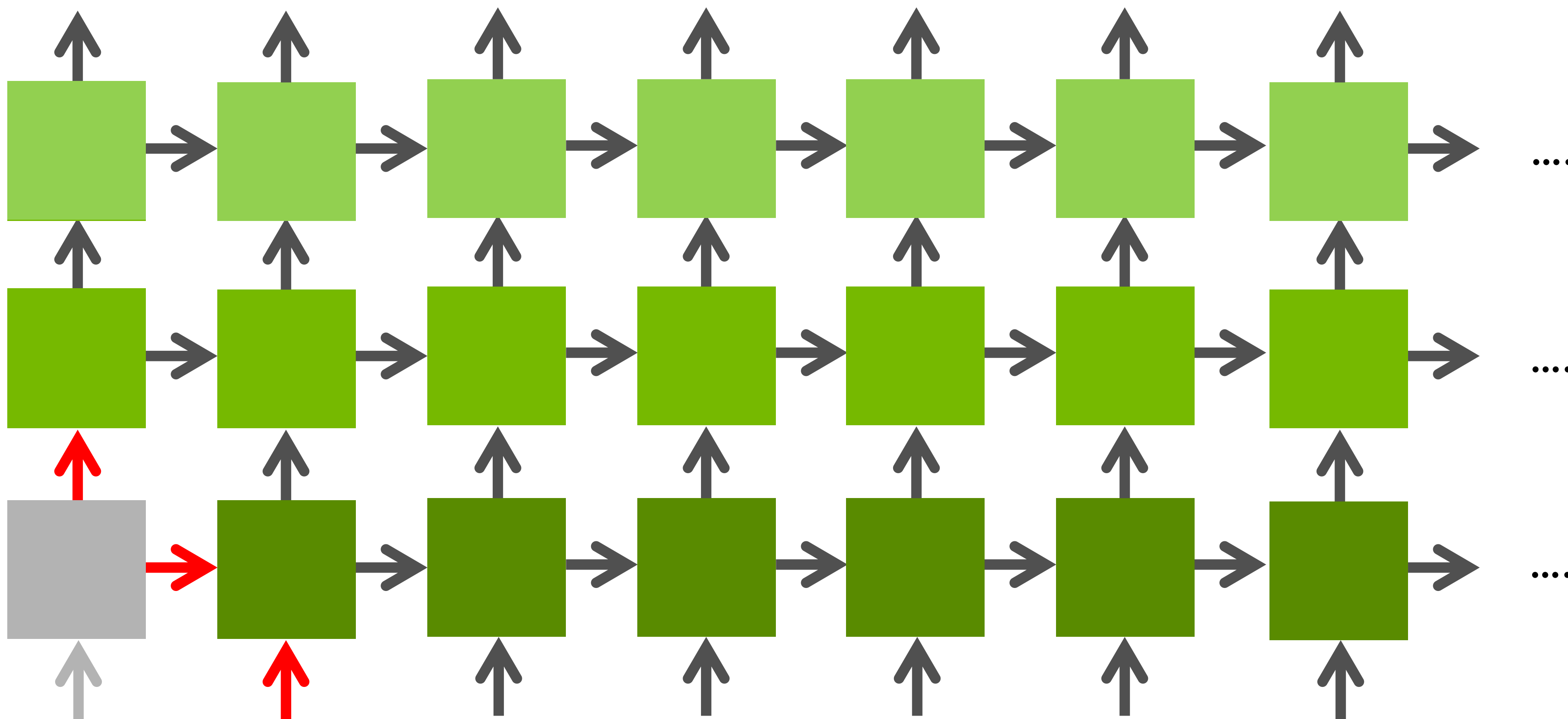
Optimization #3



NVIDIA

Increasing Parallelism

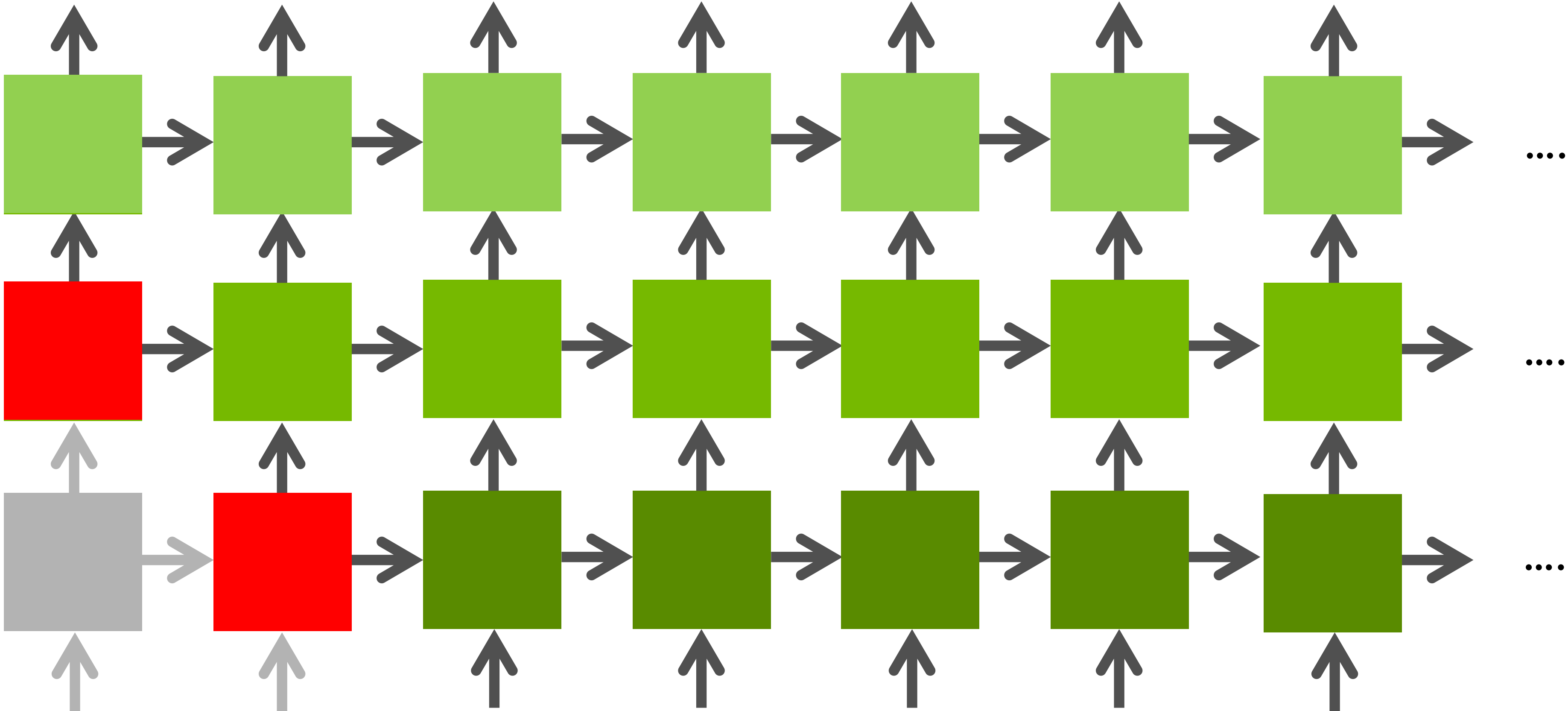
Optimization #3



NVIDIA.

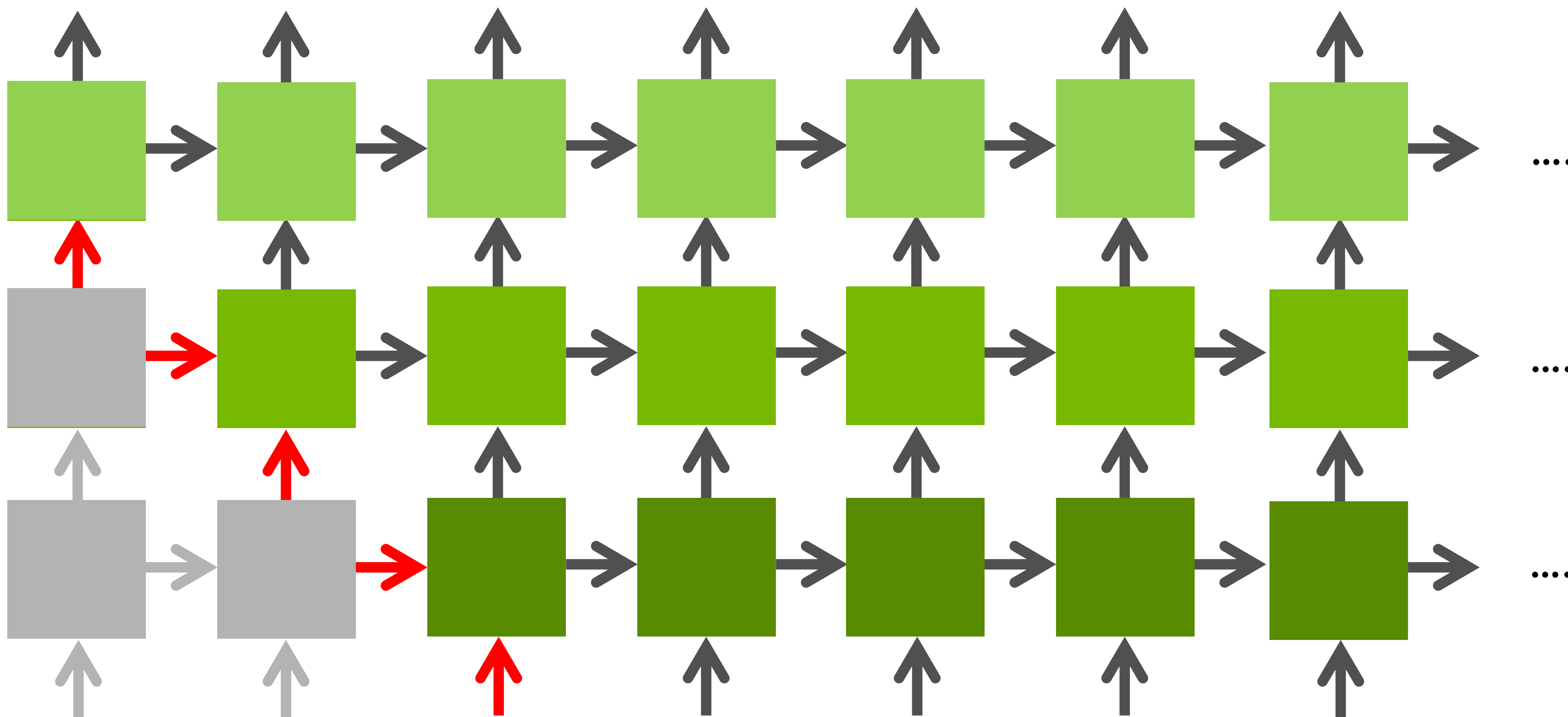
Increasing Parallelism

Optimization #3



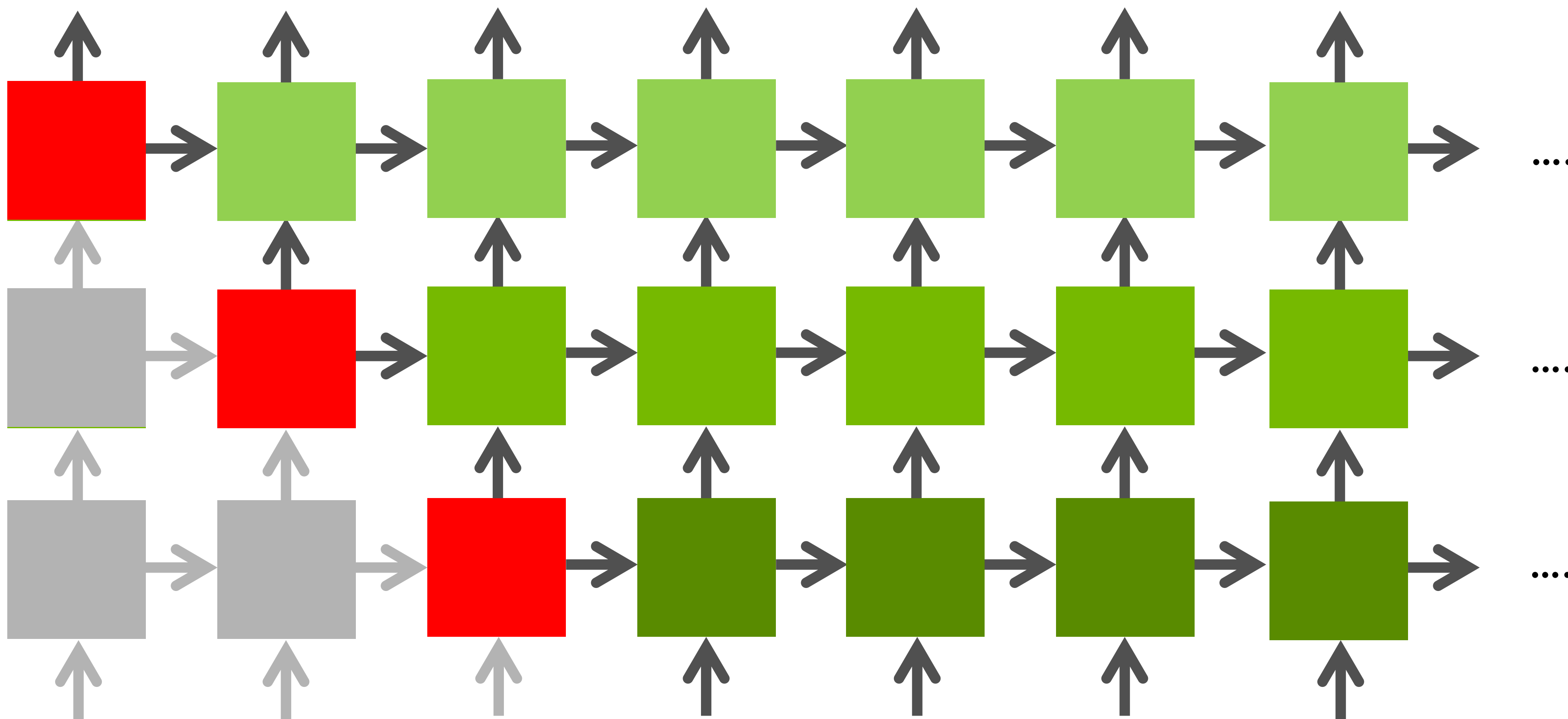
Increasing Parallelism

Optimization #3



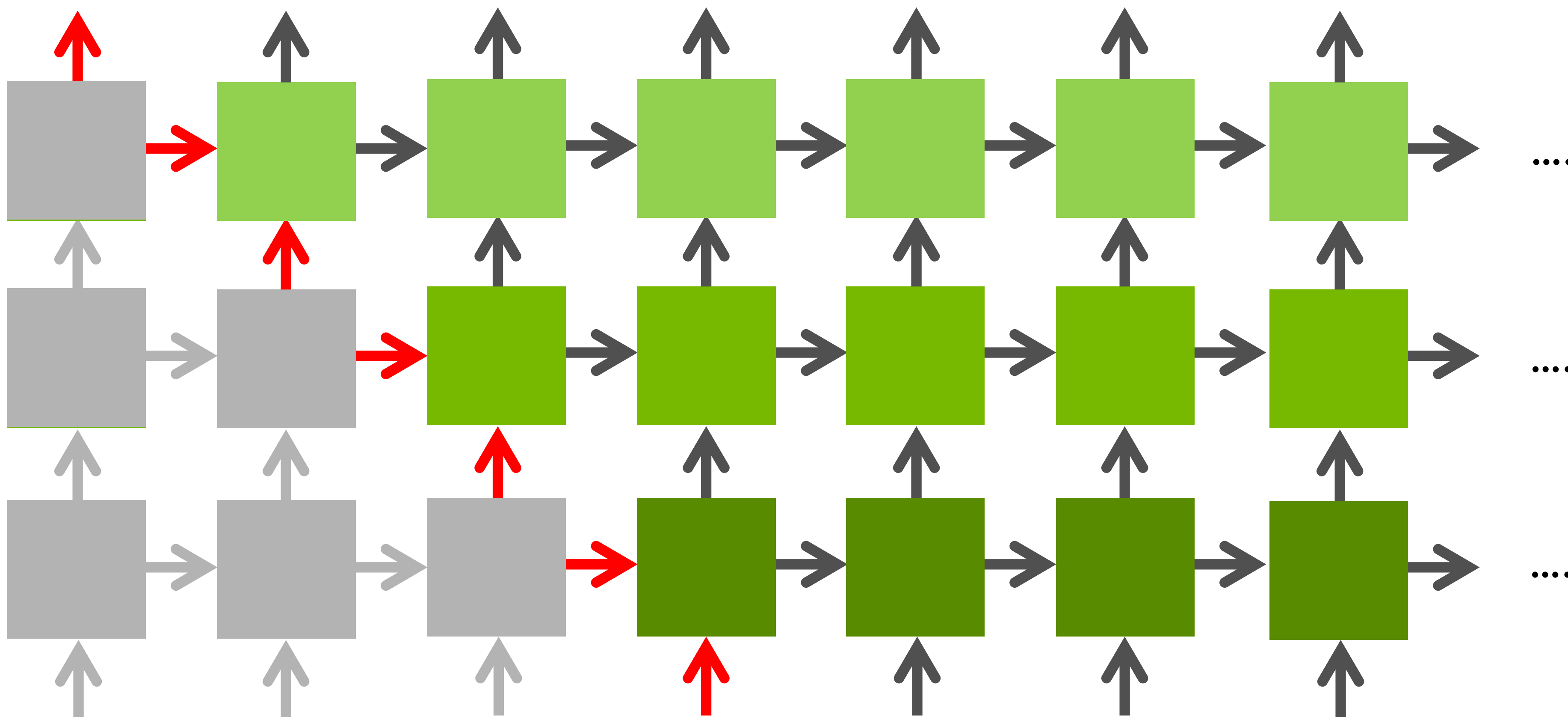
Increasing Parallelism

Optimization #3



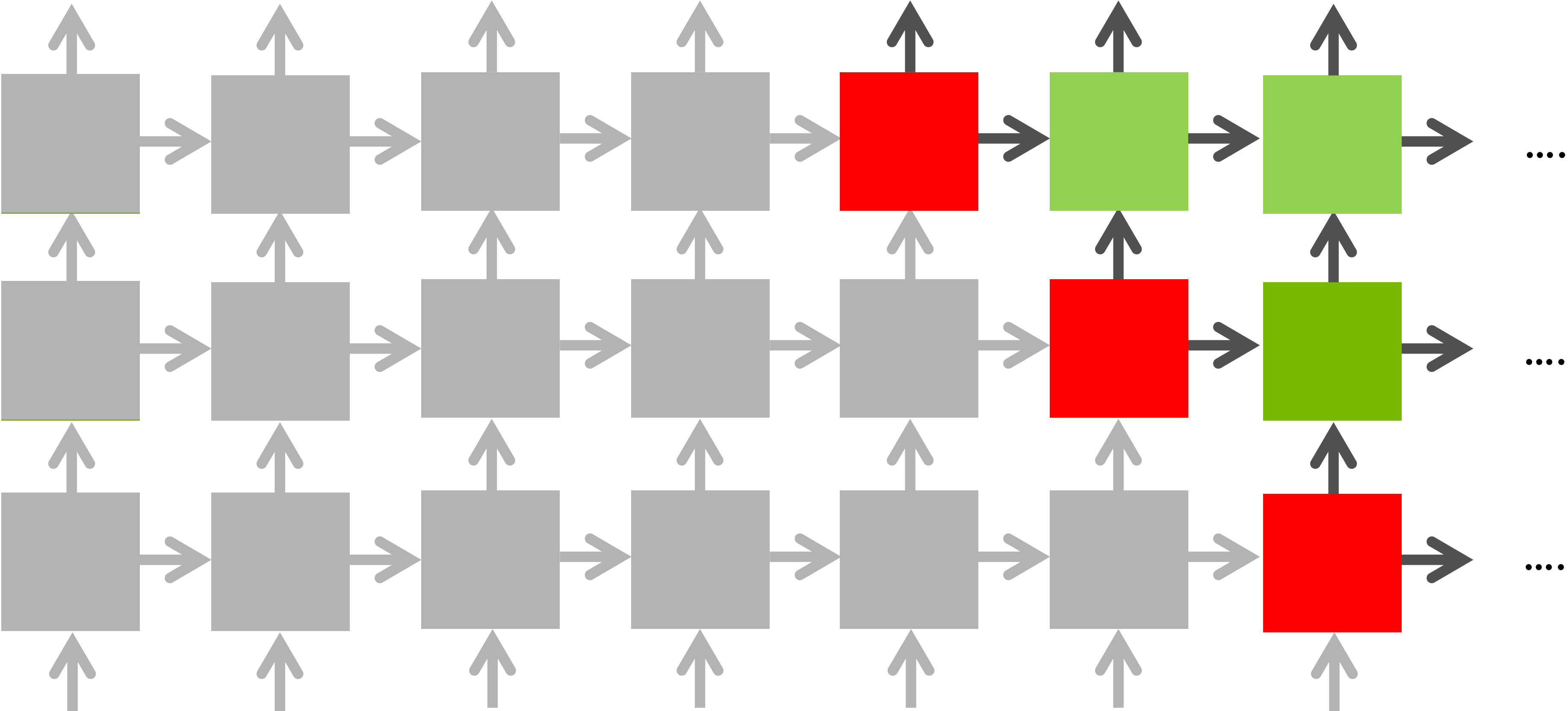
Increasing Parallelism

Optimization #3



Increasing Parallelism

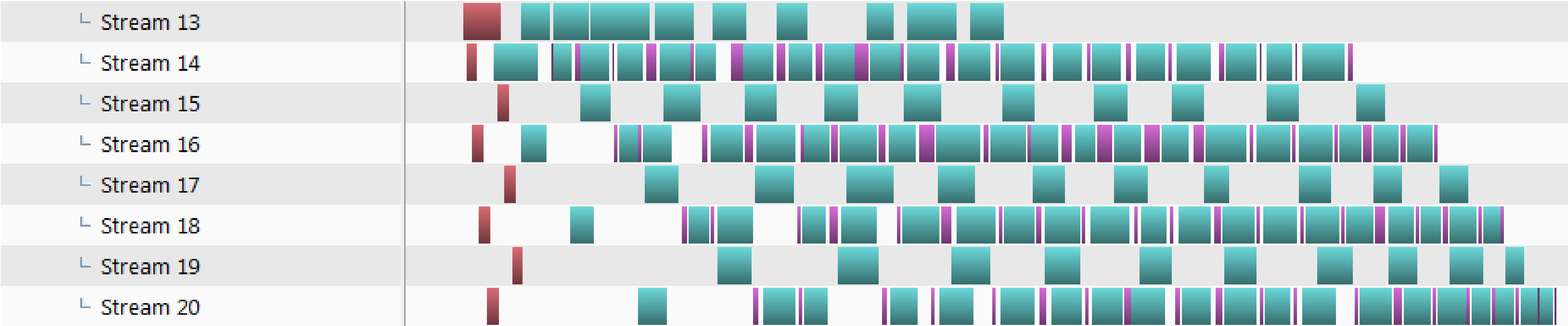
Optimization #3



Performance

Using Streams + Layers

Output of the NVIDIA visual profiler:



Three Optimizations

Speedup

For an LSTM with the following properties:

- 512 hidden units
- 100 recurrent iterations
- Minibatch 64
- Four layers

Before: 83.6ms/pass

After: 23.8ms/pass

cuDNN

Library for Neural Networks

NVIDIA provides a free library for accelerating Neural Network operations: cuDNN

cuDNN is integrated into all major frameworks and provides optimized routines for many neural network architectures, including basic RNNs, GRUs and LSTMs.

More info and download here: developer.nvidia.com/cudnn

Other libraries are available for BLAS operations, FFT, random number generation, and many more operations.

Final Words

Both hardware and software choices can greatly reduce time to solution.

It’s important to be aware of performance trade-offs being made when designing and executing a network.

Libraries and frameworks are intended to do as much of this as possible for you, but it may be you have to do a little extra work to get best performance, especially when straying off the beaten path

NVIDIA