A peer-reviewed version of this preprint was published in PeerJ on 24 July 2017.

View the peer-reviewed version (peerj.com/articles/cs-127), which is the preferred citable publication unless you specifically need to cite this preprint.

Mitchell R, Frank E. 2017. Accelerating the XGBoost algorithm using GPU computing. PeerJ Computer Science 3:e127
https://doi.org/10.7717/peerj-cs.127

Accelerating the XGBoost algorithm using GPU computing

Rory Mitchell Corresp., [1] , **Eibe Frank** Corresp. [1]

[1] Department of Computer Science, University of Waikato, Hamilton, New Zealand

Corresponding Authors: Rory Mitchell, Eibe Frank
Email address: ramitchellnz@gmail.com, eibe@cs.waikato.ac.nz

We present a CUDA based implementation of a decision tree construction algorithm within the gradient boosting library XGBoost. The tree construction algorithm is executed entirely on the GPU and shows high performance with a variety of datasets and settings, including sparse input matrices. Individual boosting iterations are parallelized, combining two approaches. An interleaved approach is used for shallow trees, switching to a more conventional radix sort based approach for larger depths. We show speedups of between 3-6x using a Titan X compared to a 4 core i7 CPU, and 1.2x using a Titan X compared to 2x Xeon CPUs (24 cores). We show that it is possible to process the Higgs dataset (10 million instances, 28 features) entirely within GPU memory. The algorithm is made available as a plug-in within the XGBoost library and fully supports all XGBoost features including classification, regression and ranking tasks.

Accelerating the XGBoost Algorithm Using GPU Computing

Rory Mitchell[1] **and Eibe Frank**[2]

[1,2]**Department of Computer Science, University of Waikato**

Corresponding author:
Rory Mitchell

Email address: ramitchellnz@gmail.com

ABSTRACT

We present a CUDA based implementation of a decision tree construction algorithm within the gradient boosting library XGBoost. The tree construction algorithm is executed entirely on the GPU and shows high performance with a variety of datasets and settings, including sparse input matrices. Individual boosting iterations are parallelized, combining two approaches. An interleaved approach is used for shallow trees, switching to a more conventional radix sort based approach for larger depths. We show speedups of between 3-6x using a Titan X compared to a 4 core i7 CPU, and 1.2x using a Titan X compared to 2x Xeon CPUs (24 cores). We show that it is possible to process the Higgs dataset (10 million instances, 28 features) entirely within GPU memory. The algorithm is made available as a plug-in within the XGBoost library[*] and fully supports all XGBoost features including classification, regression and ranking tasks.

1 INTRODUCTION

Gradient boosting is an important tool in the field of supervised learning, providing state of the art performance on classification, regression and ranking tasks. XGBoost is an implementation of a generalised gradient boosting algorithm that has become a tool of choice in machine learning competitions. This is due to its excellent predictive performance, highly optimised multicore and distributed machine implementation and the ability to handle sparse data.

Despite good performance relative to existing gradient boosting implementations, XGBoost can be very time consuming to run. Common tasks can take hours or even days to complete. Building highly accurate models using gradient boosting also requires extensive parameter tuning. In this process the algorithm must be run many times to explore the effect of parameters such as the learning rate and L1/L2 regularisation terms on cross validation accuracy. By far the most time-consuming part of the XGBoost algorithm is the construction of decision trees within each boosting iteration. This paper describes and evaluates a GPU algorithm for accelerating decision tree construction within individual boosting iterations in the single machine XGBoost setting.

Graphics processing units (GPUs) have recently been used to accelerate compute intensive tasks in machine learning and many other fields through the utilisation of their specialised SIMD architecture. We show that GPUs are an effective tool for accelerating the gradient boosting process and can provide significant speed advantages.

GPU accelerated decision tree algorithms have been tried before with moderate success. Our unique contributions are as follows. We describe a completely GPU based implementation that scales to arbitrary numbers of leaf nodes and exhibits stable performance characteristics on a range of datasets and settings. We experiment with novel approaches to processing interleaved subsets of data on GPUs and develop a massively parallel tree construction algorithm that natively handles sparse data. We also provide a feature complete implementation for classification, regression and learning to rank tasks in the open source XGBoost library.[1]

[1] https://github.com/dmlc/xgboost/tree/master/plugin/updater_gpu

2 BACKGROUND AND RELATED WORK

We review the basic strategy of tree boosting for machine learning and revisit the derivation of the XGBoost algorithm, before considering the execution model and memory architecture of GPUs as well as languages and libraries for GPU computing. Our GPU-based implementation makes extensive use of high-performance GPU primitives and we discuss these next. We briefly discuss the effect of using single-precision floating point arithmetic before reviewing related work on GPU-based induction of decision trees from data.

2.1 Tree Boosting Algorithms

XGBoost is a supervised learning algorithm that implements a process called boosting to yield accurate models. Supervised learning refers to the task of inferring a predictive model from a set of labelled training examples. This predictive model can then be applied to new unseen examples. The inputs to the algorithm are pairs of training examples $(\vec{x_0}, y_0), (\vec{x_1}, y_1) ... (\vec{x_n}, y_n)$ where $\vec{x}$ is a vector of features describing the example and y is its label. Supervised learning can be thought of as learning a function $F(\vec{x}) = y$ that will correctly label new input instances.

Supervised learning may be used to solve classification or regression problems. In classification problems the label y takes a discrete (categorical) value. For example we may wish to predict if a manufacturing defect occurs or does not occur based on attributes recorded from the manufacturing process, such as temperature or time, that are represented in $\vec{x}$. In regression problems the target label y takes a continuous value. This can be used to frame a problem such as predicting temperature or humidity on a given day.

XGBoost is at its core a decision tree boosting algorithm. Boosting refers to the ensemble learning technique of building many models sequentially, with each new model attempting to correct for the deficiencies in the previous model. In tree boosting each new model that is added to the ensemble is a decision tree. We explain how to construct a decision tree model and how this can be extended to generalised gradient boosting with the XGBoost algorithm.

2.1.1 Decision Trees

Decision tree learning is a method of predictive modelling that learns a model by repeatedly splitting subsets of the training examples (also called *instances*) according to some criteria. Decision tree inducers are supervised learners that accept labelled training examples as an input and generate a model that may be used to predict the labels of new examples.

In order to construct a decision tree we start with the full set of training instances and evaluate all possible ways of creating a binary split among those instances based on the input features in $\vec{x}$. We choose the split that produces the most meaningful separation of the target label y. Different measures can be used to evaluate the quality of a split. After finding the ”best” split we can create a node in the tree that partitions training instances down the left or right branch according to some feature value. The subsets of training instances can then be recursively split to continue growing the tree to some maximum depth or until the quality of the splits is below some threshold. The leaves of the tree will contain predictions for the target label y. For categorical labels the prediction can be set as the majority class from the training instances that end up in that leaf. For regression tasks, the label prediction can be set as the mean of the training instances in that leaf.

To use the tree for prediction we can input an unlabelled example at the root of the tree and follow the decision rules until the example reaches a leaf. The unlabelled example can be labelled according to the prediction of that leaf.

Figure 1 shows an example decision tree that can predict whether or not an individual owns a house. The decision is based on their age and whether or not they have a job. The tree correctly classifies all instances from Table 1.

Decision tree algorithms typically expand nodes from the root in a greedy manner in order to maximise some criterion measuring the value of the split. For example, decision tree algorithms from the C4.5 family (Quinlan, 2014), designed for classification, use information gain as the split criterion. Information gain describes a change in entropy H from some previous state to a new state. Entropy is defined as

$$H(T) = -\sum_{y \in Y} \mathrm{P}(y) \log_b \mathrm{P}(y)$$

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Instance	Age	Has job	Owns house
0	12	N	N
1	32	Y	Y
2	25	Y	Y
3	48	N	N
4	67	N	Y
5	18	Y	N

Table 1. Example training instances

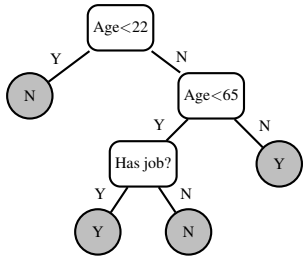

Figure 1. Example decision tree

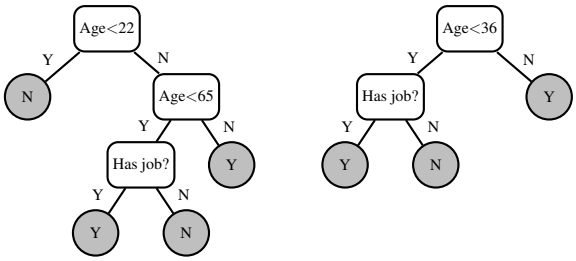

Figure 2. Decision Tree Ensemble

Where T is a set of labelled training instances, $y \in Y$ is an instance label and $P(y)$ is the probability of drawing an instance with label y from T. Information gain is defined as

$$IG(T, T_{left}, T_{right}) = H_T - (n_{left}/n_{total}) * H(T_{left}) - (n_{right}/n_{total}) * H(T_{right})$$

Here T_{left} and T_{right} are the subsets of T created by a decision rule. n_{total}, n_{left} and n_{right} refer to the number of examples in the respective sets.

Many different criteria exist for evaluating the quality of a split. Any function can be used that produces some meaningful separation of the training instances with respect to the label being predicted.

In order to find the split that maximises our criterion we can enumerate all possible splits on the input instances for each feature. In the case of numerical features and assuming the data has been sorted, this enumeration can be performed in $O(nm)$ steps, where n is the number of instances and m is the number of features. A scan is performed from left to right on the sorted instances, maintaining a running sum of labels as the input to the gain calculation. We do not consider the case of categorical features in this paper because XGBoost encodes all categorical features using one hot encoding and transforms them into numerical features.

Another consideration when building decision trees is applying some form of regularisation to prevent overfitting. Overfitting on training data leads to poor model generalisation ability and poor performance on test data. Given a sufficiently large decision tree it is possible to generate unique decision rules for every instance in the training set such that each training instance is correctly labelled. This results in 100% accuracy on the training set but may perform poorly on new data. For this reason it is necessary to limit the growth of the tree during construction or apply pruning after construction.

2.1.2 Gradient Boosting

Decision trees produce easy to interpret models useful for a variety of problems, but their accuracy can be considerably improved when many trees are combined into an ensemble learner. For example, given an input instance to be classified, we can test it against many trees built on different subsets of the training set and return the mode of all predictions. This has the effect of reducing classifier error because it reduces variance in the estimate of the classifier.

Figure 2 shows an ensemble of two decision trees. We can predict the output label using all trees by taking the most common class prediction or some weighted average of all predictions.

Ensemble learning methods can also be used to reduce the bias component in the classification error of the base learner. Boosting is an ensemble method that creates ensemble members sequentially. The newest member is created to compensate for the instances incorrectly labelled by the previous learners.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Gradient boosting is a variation on boosting which represents the learning problem as gradient descent on some arbitrary differentiable loss function that measures the performance of the model on the training set. More specifically, the boosting algorithm executes M boosting iterations to learn a function $F(x)$ that outputs predictions $\hat{y} = F(x)$ minimising some loss function $L(y, \hat{y})$. At each iteration we add a new estimator $f(x)$ to try to correct the prediction of y for each training instance.

$$F_{m+1}(x) = F_m(x) + f(x) = y$$

We can correct the model by setting $f(x)$ to:

$$f(x) = y - F_m(x)$$

This fits the model $f(x)$ for the current boosting iteration to the residuals $y - F_m(x)$ of the previous iteration. In practice, we approximate $f(x)$, for example by using a depth limited decision tree.

This iterative process can be shown to be a gradient descent algorithm when the loss function is the squared error:

$$L(y, F(x)) = \frac{1}{2}(y - F(x))^2$$

The loss over all training instances can be written as

$$J = \sum_i L(y_i, F(x_i))$$

We seek to minimise J by adjusting $F(x_i)$. The gradient for a particular instance x_i is given by

$$\frac{dJ}{dF(x_i)} = \frac{d\sum_i L(y_i, F(x_i))}{dF(x_i)} = \frac{dL(y_i, F(x_i))}{dF(x_i)} = F_m(x_i) - y_i$$

We can see that the residuals are the negative gradient of the squared error loss function:

$$f(x) = y - F_m(x) = -\frac{dL(y, F(x))}{dF(x)}$$

By adding a model that approximates this negative gradient to the ensemble we move closer to a local minimum of the loss function, thus implementing gradient descent.

2.1.3 Generalised Gradient Boosting and XGBoost

Here we derive the XGBoost algorithm following the explanation in (Chen and Guestrin, 2016). XGBoost is a generalized gradient boosting implementation that includes a regularisation term, used to combat overfitting, as well as support for arbitrary differentiable loss functions.

Instead of optimising plain squared error loss, an objective function with two parts is defined, a loss function over the training set as well as a regularisation term which penalises the complexity of the model:

$$Obj = \sum_i L(y_i, \hat{y}_i) + \sum_k \Omega(f_k)$$

$L(y_i, \hat{y}_i)$ can be any convex differentiable loss function that measures the difference between the prediction and true label for a given training instance. $\Omega(f_k)$ describes the complexity of tree f_k and is defined in the XGBoost algorithm (Chen and Guestrin, 2016) as

$$\Omega(f_k) = \gamma T + \frac{1}{2}\lambda w^2 \tag{1}$$

where T is the number of leaves of tree f_k and w are the leaf weights (i.e., the predicted values stored at the leaf nodes). When $\Omega(f_k)$ is included in the objective function we are forced to optimize for a less complex tree that simultaneously minimizes $L(y_i, \hat{y}_i)$. This helps to reduce overfitting. γT provides a constant penalty for each additional tree leaf and λw^2 penalises extreme weights. γ and λ are user configurable parameters.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Given that boosting proceeds in an iterative manner we can state the objective function for the current iteration m in terms of the prediction of the previous iteration $\hat{y_i}^{(m-1)}$ adjusted by the newest tree f_k:

$$Obj^m = \sum_i L(y_i, \hat{y_i}^{(m-1)} + f_k(x_i)) + \sum_k \Omega(f_k)$$

We can then optimise to find the f_k which minimises our objective.

Taking the Taylor expansion of the above function to the second order allows us to easily accommodate different loss functions:

$$Obj^m \simeq \sum_i [L(y_i, \hat{y_i}^{(m-1)}) + g_i f_k(x) + \frac{1}{2} h_i f_k(x)^2] + \sum_k \Omega(f_k) + constant$$

Here, g_i and h_i are the first and second order derivatives respectively of the loss function for instance i:

$$g_i = \frac{dL(y_i, \hat{y_i}^{(m-1)})}{d\hat{y_i}^{(m-1)}} \quad h_i = \frac{d^2 L(y_i, \hat{y_i}^{(m-1)})}{d(\hat{y_i}^{(m-1)})^2}$$

Note that the model $\hat{y_i}^{(m-1)}$ is left unchanged during this optimisation process. The simplified objective function with constants removed is

$$Obj^m = \sum_i [g_i f_k(x) + \frac{1}{2} h_i f_k(x)^2] + \sum_k \Omega(f_k)$$

We can also make the observation that a decision tree predicts constant values within a leaf. $f_k(x)$ can then be represented as $w_{q(x)}$ where w is the vector containing scores for each leaf and $q(x)$ maps instance x to a leaf.

The objective function can then be modified to sum over the tree leaves and the regularization term from Equation 1:

$$Obj^m = \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_{q(x)} + \frac{1}{2} (\sum_{i \in I_j} h_i) w_{q(x)}^2] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^{T} w^2$$

Here, I_j refers to the set of training instances in leaf j. The sums of the derivatives in each leaf can be defined as follows:

$$G_j = \sum_{i \in I_j} g_i \quad H_j = \sum_{i \in I_j} h_i$$

Also note that $w_{q(x)}$ is a constant within each leaf and can be represented as w_j. Simplifying we get

$$Obj^m = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2] + \gamma T \tag{2}$$

The weight w_j for each leaf minimises the objective function at

$$\frac{\partial Obj^m}{\partial w_j} = G_j + (H_j + \lambda) w_j = 0$$

The best leaf weight w_j given the current tree structure is then

$$w_j = -\frac{G_j}{H_j + \lambda}$$

Using the best w_j in Equation 2 the objective function for finding the best tree structure then becomes

$$Obj^m = -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \tag{3}$$

Equation 3 is used in XGBoost as a measure of the quality of a given tree.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Feature Value	0.1	0.4	0.5	0.6	0.9	1.1
g_i	0.1	0.8	0.2	-1.1	-0.2	-0.5
h_i	1.0	1.0	1.0	1.0	1.0	1.0
G_L	0.0	0.1	0.9	1.1	0.0	-0.2
H_L	0.0	1.0	2.0	3.0	4.0	5.0

Table 2. Enumerating Splits

Instance Id	f0	f1	f2
0	0.32	399	10.1
1	0.27	521	11.3
2	0.56	896	13.0
3	0.11	322	9.7

Table 3. Example Data Matrix

Instance Id	f0	f1	f2
0	1	0	0
1	1	0	0
2	0	0	1
3	0	1	0

Table 4. Sparse Data Matrix

2.1.4 Growing a tree

Given that it is intractable to enumerate through all possible tree structures we greedily expand the tree from the root node. In order to evaluate the usefulness of a given split we can look at the contribution of a single leaf node j to the objective function from Equation 3:

$$Obj_{leaf} = -\frac{1}{2}\frac{G_j^2}{H_j+\lambda} + \gamma$$

We can then consider the contribution to the objective function from splitting this leaf into two leaves:

$$Obj_{split} = -\frac{1}{2}\frac{G_{jL}^2}{H_{jL}+\lambda} + \frac{G_{jR}^2}{H_{jR}+\lambda} + 2\gamma$$

The improvement to the objective function from creating the split is then defined as

$$Gain = Obj_{split} - Obj_{leaf}$$

which yields

$$Gain = \frac{1}{2}[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}] - \gamma \quad (4)$$

The quality of any given split separating a set of training instances is evaluated using the gain function in Equation 4. The gain function represents the reduction in the objective function from Equation 3 obtained by taking a single leaf node j and partitioning it into two leaf nodes. This can be thought of as the increase in quality of the tree obtained by creating the left and right branch as compared to simply retaining the original node. This formula is applied at every possible split point and we expand the split with maximum gain. We can continue to grow the tree while this gain value is positive. The γ regularisation cost at each leaf will prevent the tree arbitrarily expanding. The split point selection is performed in $O(nm)$ time (given n training instances and m features) by scanning left to right through all feature values in a leaf in sorted order. A running sum of G_L and H_L is kept as we move from left to right, as shown in Table 2. G_R and H_R are inferred from this running sum and the node total.

Table 2 shows an example set of instances in a leaf. We can assume we know the sums G and H within this node as these are simply the G_L or G_R from the parent split. Therefore we have everything we need to evaluate *Gain* for every possible split within these instances and select the best.

2.1.5 XGBoost: Data Format

Tabular data input to a machine learning library such as XGBoost or Weka (Hall et al., 2009) can be typically described as a matrix with each row representing an instance and each column representing a feature as shown in Table 3. If f2 is the feature to be predicted then an input training pair $(\vec{x}_i, y_i)$ takes the form $((f0_i, f1_i), f2_i)$ where i is the instance id. A data matrix within XGBoost may also contain missing

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

values. One of the key features of XGBoost is the ability to store data in a sparse format by implicitly keeping track of missing values instead of physically storing them. While XGBoost does not directly support categorical variables, the ability to efficiently store and process sparse input matrices allows us to process categorical variables through one hot encoding. Table 4 shows an example where a categorical feature with three values is instead encoded as three binary features. The zeros in a one hot encoded data matrix can be stored as missing values. XGBoost users may specify values to be considered as missing in the input matrix or directly input sparse formats such as libsvm files to the algorithm.

2.1.6 XGBoost: Handling Missing Values

Representing input data using sparsity in this way has implications on how splits are calculated. XGBoost's default method of handling missing data when learning decision tree splits is to find the best "missing direction" in addition to the normal threshold decision rule for numerical values. So a decision rule in a tree now contains a numeric decision rule such as $f0 \leq 5.53$ but also a missing direction such as $missing = right$ that sends all missing values down the right branch. For a one hot encoded categorical variable where the zeros are encoded as missing values this is equivalent to testing "one vs all" splits for each category of the categorical variable.

The missing direction is selected as the direction which maximises the gain from Equation 4. When enumerating through all possible split values we can also test the effect on our gain function of sending all missing examples down the left or right branch and select the best option. This is slightly complicated to implement in practice as we do not know the gradient statistics of the missing values for any given feature we are working on, although we do know the sum of all the gradient statistics for the current node. The XGBoost algorithm handles this by performing two scans over the input data, the second being in the reverse direction. In the first left to right scan the gradient statistics for the left direction are the scan values maintained by the scan, the gradient statistics for the right direction are the sum gradient statistics for this node minus the scan values. Hence, the right direction implicitly includes all of the missing values. When scanning from right to left the reverse is true and the left direction includes all of the missing values. The algorithm then selects the best split from either the forwards or backwards scan.

2.2 Graphics Processing Units

The purpose of this paper is to describe how to efficiently implement decision tree learning for XGBoost on a GPU. GPUs can be thought of at a high level as having a shared memory architecture with multiple SIMD (single instruction multiple data) processors. These SIMD processors operate in lockstep typically in batches of 32 "threads" (Matloff, 2011). GPUs are optimised for high throughput and work to hide latency through the use of massive parallelism. This is in contrast to CPUs which use multiple caches, branch prediction and speculative execution in order to optimize latency with regards to data dependencies (Baxter, 2013). GPUs have been used to accelerate a variety of tasks traditionally run on CPUs, providing significant speedups for parallelizable problems with a high arithmetic intensity. Of particular relevance to machine learning is the use of GPUs to train extremely large neural networks. It was shown in 2013 that 1 billion parameter networks could be trained in a few days on three GPU machines (Coates et al., 2013). However, to the best of our knowledge, GPUs have not been used for gradient boosting previously.

2.2.1 Languages and Libraries

The two main languages for general purpose GPU programming are CUDA and OpenCL. CUDA was chosen for the implementation discussed in this paper due to the availability of optimised and production ready libraries. The GPU tree construction algorithm would not be possible without a strong parallel primitives library. We make extensive use of scan, reduce and radix sort primitives from the CUB (Merrill and NVIDIA-Labs, 2016) and Thrust (Hoberock and Bell, 2017) libraries. These parallel primitives are described in detail in Section 2.3. The closest equivalent to these libraries in OpenCL is the Boost Compute library. Several bugs were encountered when attempting to use Boost Compute and the performance of its sorting primitives lagged considerably behind CUB/Thrust. At the time of writing this paper OpenCL was not a practical option for this type of algorithm.

2.2.2 Execution model

CUDA code is written as a kernel to be executed by many thousands of threads. All threads execute the same kernel function but their behaviour may be distinguished through a unique thread ID. Listing 1

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Listing 1. Example CUDA Kernel

```
__global__ void example(float *d_a, float *d_b,
  float *d_output, int n){

  int global_tid = blockIdx.x * blockDim.x + threadIdx.x;

  if(global_tid < n){
    d_output[global_tid] = d_a[global_tid] + d_b[global_tid];
  }
}
```

shows an example kernel adding values from two arrays into an output array. Indexing is determined by the global thread ID and any unused threads are masked off with a branch statement.

Threads are grouped according to thread blocks that typically each contain some multiple of 32 threads. A group of 32 threads is known as a warp. Thread blocks are queued for execution on hardware streaming multiprocessors. Streaming multiprocessors switch between different warps within a block during program execution in order to hide latency. Global memory latency may be hundreds of cycles and as such it is important to launch sufficiently many warps within a thread block to facilitate latency hiding.

A thread block provides no guarantees about the order of thread execution unless explicit memory synchronization barriers are used. Synchronisation across thread blocks is not generally possible within a single kernel launch. Device wide synchronization is achieved by multiple kernel launches. For example, if a global synchronisation barrier is required within a kernel, the kernel must be separated into two distinct kernels where synchronisation occurs between the kernel launches.

2.2.3 Memory architecture

CUDA exposes three primary tiers of memory for reading and writing. Device wide global memory, thread block accessible shared memory and thread local registers.

- **Global memory** Global memory is accessible by all threads and has the highest latency. Input data, output data and large amounts of working memory are typically stored in global memory. Global memory can be copied from the device (i.e., the GPU) to the host computer and vice versa. Bandwidth of host/device transfers is much slower than that of device/device transfers and should be avoided if possible. Global memory is accessed in 128 byte cache lines on current GPUs. Memory accesses should be coalesced in order to achieve maximum bandwidth. Coalescing refers to the grouping of aligned memory load/store operations into a single transaction. For example, a fully coalesced memory read occurs when a warp of 32 threads loads 32 contiguous 4 byte words (128 bytes). Fully uncoalesced reads (typical of gather operations) can limit device bandwidth to less than 10% of peak bandwidth (Harris, 2013).

- **Shared memory** 48KB of shared memory is available to each thread block. Shared memory is accessible by all threads in the block. Shared memory has a significantly lower latency than global memory and is typically used as working storage within a thread block. It is sometimes described as a ”programmer managed cache”.

- **Registers** A finite number of local registers is available to each thread. Operations on registers are generally the fastest. Threads within the same warp may read/write registers from other threads in the warp via intrinsic instructions such as shuffle or broadcast (Nvidia, 2017).

2.3 Parallel Primitives

GPU primitives are small algorithms used as building blocks in massively parallel algorithms. While many data parallel tasks can be expressed with simple programs without them, GPU primitives may be used to compose more complicated algorithms while retaining high performance, readability and reliability. Understanding which specific tasks can be achieved using parallel primitives and the relative performance of GPU primitives as compared to their CPU counterparts is key to designing effective GPU algorithms.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

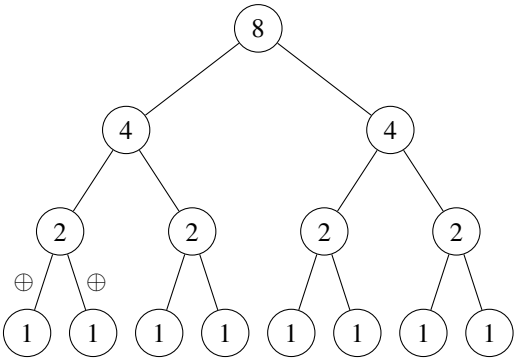

Figure 3. Sum parallel reduction

Listing 2. Warp Reduction

```
__device__
float warp_reduce(float x) {
  for (int d = 16; d > 0; d /= 2)
    x += __shfl_down(x, d);
  return x;
}
```

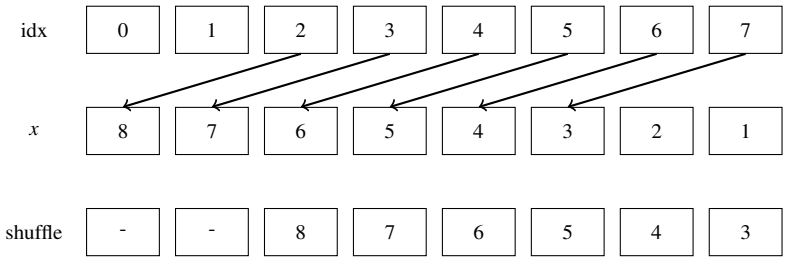

Figure 4. Shuffle down intrinsic: $d = 2$

2.3.1 Reduction

A parallel reduction reduces an array of values into a single value using a binary associative operator. Given a binary associative operator $\oplus$ and an array of elements the reduction returns $(a_0 \oplus a_1 \oplus ... \oplus a_{n-1})$. Note that floating point addition is not strictly associative. This means a sequential reduction operation will likely result in a different answer to a parallel reduction (the same applies to the scan operation described below). This is discussed in greater detail in Section 2.5. The reduction operation is easy to implement in parallel by passing partial reductions up a tree, taking $O(\log n)$ iterations given n input items and n processors. This is illustrated in Figure 3.

In practice, GPU implementations of reductions do not launch one thread per input item but instead perform parallel reductions over "tiles" of input items then sum the tiles together sequentially. The size of a tile varies according to the optimal granularity for a given hardware architecture. Reductions are also typically tiered into three layers: warp, block and kernel. Individual warps can very efficiently perform partial reductions over 32 items using shuffle instructions introduced from Nvidia's Kepler GPU architecture onwards. The thread block can then combine these warp reductions to complete a tile of input. The thread block can iterate over many input tiles sequentially, summing the reduction from each. When all thread blocks are finished the results from each are summed together at the kernel level to produce the final output. Listing 2 shows code for a fast warp reduction using shuffle intrinsics to communicate between threads in the same warp. The 'shuffle down' instruction referred to in Listing 2 simply allows the current thread to read a register value from the thread d places to the left, so long as that thread is in the same warp. An example of the shuffle down function is shown in Figure 4 where each thread gathers the item $d = 2$ places to its left. The complete warp reduction algorithm requires 5 iterations to sum over 32 items.

Reductions are highly efficient operations on GPUs. An implementation is given in (Harris, 2007) that approaches the maximum bandwidth of the device tested.

2.3.2 Parallel Prefix Sum (Scan)

The prefix sum takes a binary associative operator (most commonly addition) and applies it to an array of elements. Given a binary associative operator $\oplus$ and an array of elements the prefix sum returns $[a_0, (a_0 \oplus a_1), ..., (a_0 \oplus a_1, ..., a_{n-1})]$. A prefix sum is an example of a calculation which seems inherently serial but has an efficient parallel algorithm: the Blelloch scan algorithm.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Algorithm 1: Simple scan

```
for d=1 to log2 n do
    for k=0 to n-1 in parallel do
        if k ≥ 2^(d−1) then
            x[k] := x[k − 2^(d−1)] + x[k]
        end
    end
end
```

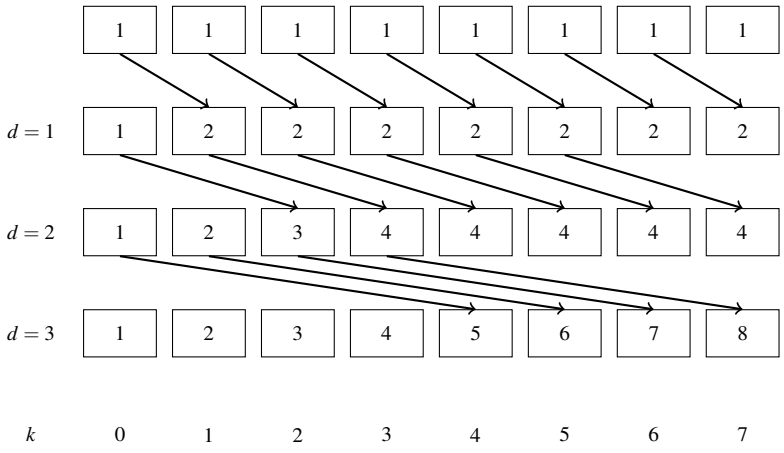

Figure 5. Simple Parallel Scan Example

Listing 3. Warp Scan

```
__device__
float warp_scan(float x) {
        int lane_id = threadIdx.x % 32;
        for (int d = 1; d < 32; d *= 2){
                float tmp = __shfl_up(x, d);
                if (lane_id >= offset){
                        x += tmp;
                }
        }
        return x;
}
```

Let us consider a simple implementation of a parallel scan first, as described in (Hillis and Steele Jr, 1986). It is given in Algorithm 1. Figure 5 shows it in operation: we apply a simple scan with the addition operator to an array of 1's. Given one thread for each input element the scan takes $\log_2 n = 3$ iterations to complete. The algorithm performs $O(n \log_2 n)$ addition operations.

Given that a sequential scan performs only n addition operations the simple parallel scan is not work efficient. A work efficient parallel algorithm will perform the same number of operations as the sequential algorithm and may provide significantly better performance in practice. A work efficient algorithm is described in (Blelloch, 1990). The algorithm is separated into two phases, an "upsweep" phase similar to a reduction and a "downsweep" phase . Pseudocode for the upsweep (Algorithm 2) and downsweep (Algorithm 3) phases are given below, following the implementation in (Harris et al., 2007).

Figures 6 and 7 show examples of the work efficient Blelloch scan, as an exclusive scan (the sum for a given item excludes the item itself). Solid lines show summation with the previous item in the array,

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Algorithm 2: Blelloch Scan - Upsweep

```
1  offset = 1
2  for d= log2 n to 1 do
3      for k=0 to n-1 in parallel do
4          if k < 2^(d-1) then
5              ai = offset × (2 × k + 1) - 1
6              bi = offset × (2 × k + 2) - 1
7              x[bi] = x[bi] + x[ai]
8          end
9      end
10     offset = offset * 2
11 end
```

Algorithm 3: Blelloch Scan - Downsweep

```
1  offset = 2^(log2 n-1)
2  x[n - 1] := 0
3  for d=1 to log2n do
4      for k=0 to n-1 in parallel do
5          if k < 2^(d-1) then
6              ai = offset × (2 × k + 1) - 1
7              bi = offset × (2 × k + 2) - 1
8              t = x[ai]
9              x[ai] = x[bi]
10             x[bi] = x[bi] + t
11         end
12     end
13     offset = offset/2
14 end
```

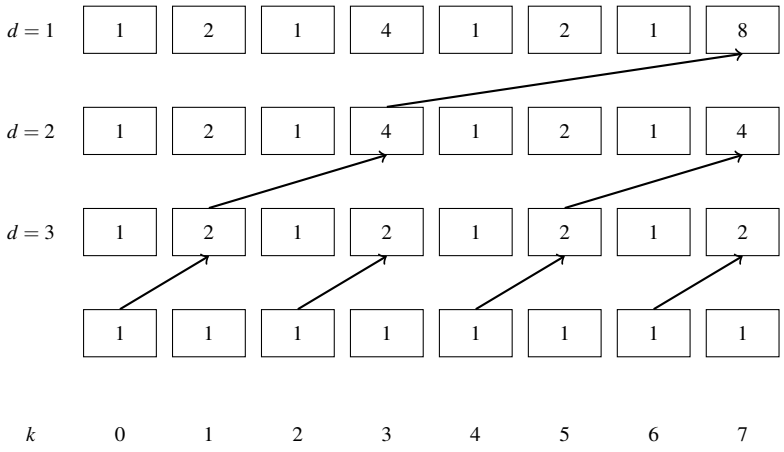

Figure 6. Blelloch Scan Upsweep Example

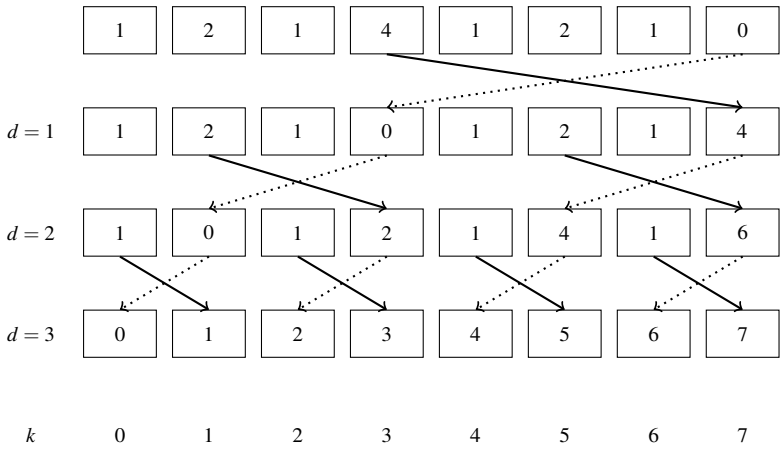

Figure 7. Blelloch Scan Downsweep Example

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Algorithm 4: Radix Sort Pass

```
Input  : X
Output : Y
for i = 0 to n - 1 in parallel do
    F[i] := bit_flip(X[i])
end
S := exclusive_scan(F)
r := S[n - 1] + F[n - 1]
for i = 0 to n - 1 in parallel do
    if X[i] = 0 then
        A[i] := S[i]
    else if X[i] = 1 then
        A[i] := i - S[i] + r
end
for i = 0 to n - 1 in parallel do
    Y[A[i]] := X[i]
end
```

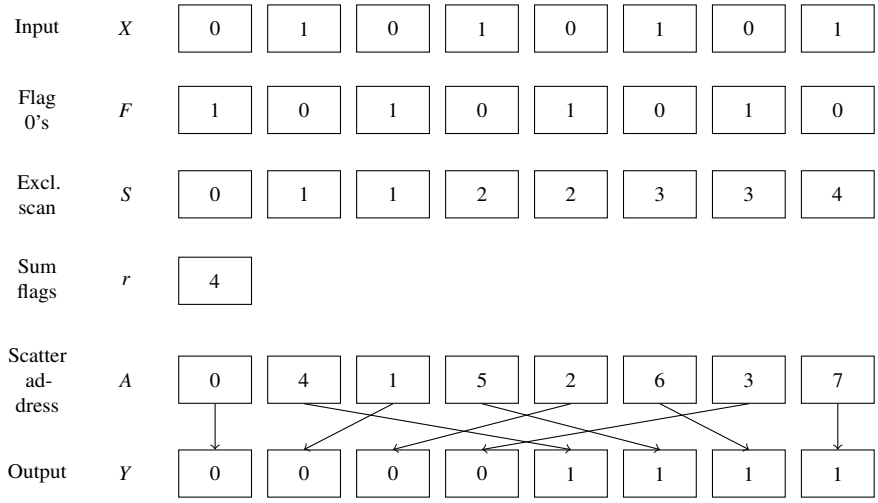

Figure 8. Radix Sort Example

dotted lines show replacement of the previous item with the new value. $O(n)$ additions are performed in both the upsweep and downsweep phase resulting in the same work efficiency as the serial algorithm.

A segmented variation of scan that processes contiguous blocks of input items with different head flags can be easily formulated. This is achieved by creating a binary associative operator on key value pairs. The operator tests the equality of the keys and sums the values if they belong to the same sequence. This is discussed further in Section 2.4.

A scan may also be implemented using warp intrinsics to create fast 32 item prefix sums based on the simple scan in Figure 5. Code for this is shown in Listing 3. Although the simple scan algorithm is not work efficient, we use this approach for small arrays of size 32.

2.3.3 Radix Sort

Radix sorting on GPUs follows from the ability to perform parallel scans. A scan operation may be used to calculate the scatter offsets for items within a single radix digit as described in Algorithm 4 and Figure 8. Flagging all ”0” digits with a one and performing an exclusive scan over these flags gives the new position of all zero digits. All ”1” digits must be placed after all ”0” digits, therefore the final positions of the ”1”s can be calculated as the exclusive scan of the ”1”s plus the total number of ”0”s. The exclusive scan of ”1” digits does not need to be calculated as it can be inferred from the array index and

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

# elements	CPU Reduce (ms)	GPU Reduce (ms)	Speedup
1024	0.001602	0.011461	0.14
32768	0.028116	0.019066	1.47
65536	0.062686	0.021943	2.86
131072	0.115125	0.019781	5.82
262144	0.235459	0.025270	9.32
524288	0.459954	0.040246	11.43
1048576	0.962114	0.049934	19.27
2097152	1.892986	0.078206	24.21
4194304	3.790061	0.128856	29.41
8388608	7.369841	0.238102	30.95
16777216	14.873887	0.452583	32.86
33554432	29.764523	0.881077	33.78
67108864	59.592416	1.742125	34.21

Table 5. GPU vs CPU Reduction Benchmark

# elements	CPU Scan (ms)	GPU Scan (ms)	Speedup
1024	0.002053	0.097941	0.02
32768	0.068269	0.087770	0.78
65536	0.135170	0.087117	1.55
131072	0.272175	0.088081	3.09
262144	0.554893	0.102699	5.40
524288	1.085465	0.125466	8.65
1048576	2.182810	0.196783	11.09
2097152	4.410221	0.300727	14.67
4194304	8.891293	0.489485	18.16
8388608	18.709616	0.867221	21.57
16777216	36.549118	1.543910	23.67
33554432	69.834813	2.912466	23.98
67108864	147.557471	5.736665	25.72

Table 6. GPU vs CPU Scan Benchmark

the exclusive scan of ”0”s. For example at index 5 (using 0 based indexing), if our exclusive scan shows a sum of 3 ”0”s, then there must be two ”1”s because a digit can only be 0 or 1.

The basic radix sort implementation only sorts unsigned integers but this can be extended to correctly sort signed integers and floating point through simple bitwise transformations. Fast implementations of GPU radix sort perform a scan over many radix bits in a single pass. Merrill and Grimshaw show a highly efficient and practical implementation of GPU radix sorting in (Merrill and Grimshaw, 2011). They show speedups of 2x over a 32 core CPU and claim to have the fastest sorting implementation for any fully programmable microarchitecture.

2.3.4 Benchmarking Parallel Primitives

Benchmarks are performed on the above parallel primitives against a single CPU thread to provide an idea of performance. These benchmarks are for context only and are not a fair comparison against many core CPUs. All operations are performed on 32-bit floats.

GPU primitives are all from the CUB library (Merrill and NVIDIA-Labs, 2016) and run on a GeForce GTX970. Times do not include transferring memory from device to host or vice versa. CPU functions are performed on a Intel i5-4590 @ 3.30GHz. The scan and reduce functions are implemented according to Listings 4 and 5. std::sort is used as the CPU sorting function. Note that this is a comparison sort with a different time complexity to radix sort – there is no standard implementation of radix sort within the C++ ecosystem. However, the comparison is nevertheless useful because std::sort is the default sorting function used in C++ code.

Listing 4. CPU Reduction

```
float sum = 0;
for(float &e : elements){
        sum += e;
}
```

Listing 5. CPU Scan

```
float tmp = 0;
for(float &e : elements){
        e += tmp;
        tmp = e;
}
```

Tables 5, 6 and 7 show that GPU primitive performance improves relative to the CPU algorithm as the input size is increased, beginning to plateau at very large sizes as the GPU becomes saturated with work. The relatively poor performance at small sizes is due to the overhead of launching GPU kernels. GPU kernel launch times are profiled in (Boyer, 2016) and found to cost between 3 and 14 microseconds. Note that the 1024 element reduction in Table 5 takes approximately 10 microseconds. At small sizes execution time is dominated by kernel launch overhead, making GPU algorithms impractical for processing small batches of data sequentially. Radix sort on the GPU still outperforms std::sort for 1024 elements, despite the small input size. This is because much more work is being done compared to scan or reduction. The kernel overhead is therefore less significant. At large sizes GPU radix sort shows dramatic performance improvements over std::sort—up to two orders of magnitude.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

# elements	CPU Sort (s)	GPU Sort (s)	Speedup
1024	0.000036	0.000024	1.48
32768	0.002116	0.000245	8.64
65536	0.004372	0.000258	16.92
131072	0.009291	0.000277	33.54
262144	0.018895	0.000414	45.62
524288	0.073453	0.000964	56.44
1048576	0.068871	0.000966	71.32
2097152	0.134228	0.001672	80.29
4194304	0.266037	0.003078	86.43
8388608	0.523203	0.005879	89.00
16777216	1.054674	0.011439	92.20
33554432	2.125154	0.022094	96.19
67108864	4.680722	0.043881	106.67

Table 7. GPU vs CPU Sort Benchmark

Sequence Id	0	0	1	0	1	1
Values	1	1	1	1	1	1
Values Scan	1	2	1	3	2	3

Table 8. Interleaved Sequences

Sequence Id	0	0	0	1	1	1
Values	1	1	1	1	1	1
Values Scan	1	2	3	1	2	3

Table 9. Segmented Sequences

Listing 6. Segmented Sum Operator

```
KeyValue op(KeyValue a, KeyValue b){
  if(a.key == b.key){
        b.value += a.value;
    return b;
  }
  else{
    return b;
  }
}
```

2.4 Scan and Reduce on Multiple Sequences

Variations on scan and reduce consider multiple sequences contained within the same input array and identified by key flags. This is useful for building decision trees as the data can be repartitioned into smaller and smaller groups as we build the tree.

We will describe an input array as containing either ”interleaved” or ”segmented” sequences. Table 8 shows an example of two interleaved sequences demarcated by flags. Its values are mixed up and do not reside contiguously in memory. This is in contrast to Table 9, with two ”segmented” sequences. The segmented sequences reside contiguously in memory.

2.4.1 Segmented Scan

A scan can be performed on the sequences from Table 9 using the conventional scan algorithm described in Section 2.3.2 and modifying the binary associative operator to accept key value pairs. Listing 6 shows an example of a binary associative operator that performs a segmented summation. It resets the sum when the key changes.

2.4.2 Segmented Reduce

A segmented reduction can be implemented efficiently by applying the segmented scan described above and collecting the final value of each sequence. This is because the last element in a scan is equivalent to a reduction.

2.4.3 Interleaved Sequences: Multireduce

A reduction operation on interleaved sequences is commonly described as a multireduce operation. To perform a multireduce using the conventional tree algorithm described in Section 2.3.1 a vector of sums can be passed up the tree instead of a single value, with one sum for each unique sequence. As the

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

GPU	Single precision	Double precision
GTX 970 (Maxwell)	3494	109
Titan X (Pascal)	10157	317

Table 10. GPU GFLOPs

Algorithm	Mean error	St.dev
Sequential	0.0694	0.0520
Parallel	0.0007	0.0005

Table 11. 32 bit Floating point precision

number of unique sequences or ”buckets” increases, this algorithm becomes impractical due to limits on temporary storage (registers and shared memory).

A multireduce can alternatively be formulated as a histogram operation using atomic operations in shared memory. Atomic operations allow multiple threads to safely read/write a single piece of memory. A single vector of sums is kept in shared memory for the entire thread block. Each thread can then read an input value and increment the appropriate sum using atomic operations. When multiple threads contend for atomic read/write access on a single piece of memory they are serialised. Therefore a histogram with only one bucket will result in the entire thread block being serialised (i.e., only one thread can operate at a time). As the number of buckets increases this contention is reduced. For this reason the histogram method will only be appropriate when the input sequences are distributed over a large number of buckets.

2.4.4 Interleaved Sequences: Multiscan

A scan operation performed on interleaved sequences is commonly described as a multiscan operation. A multiscan may be implemented, like multireduce, by passing a vector of sums as input to the binary associative operator. This increases the local storage requirements proportionally to the number of buckets.

General purpose multiscan for GPUs is discussed in (Eilers, 2014) with the conclusion that ”multiscan cannot be recommended as a general building block for GPU algorithms”. However, highly practical implementations exist that are efficient up to a limited number of interleaved buckets, where the vector of sums approach does not exceed the capacity of the device. The capacity of the device in this case refers to the amount of registers and shared memory available for each thread to store and process a vector.

Merill and Grimshaw’s optimised radix sort implementation (Merrill and NVIDIA-Labs, 2016; Merrill and Grimshaw, 2011), mentioned in Section 2.3.3, relies on an 8 way multiscan in order to calculate scatter addresses for up to 4 bits at a time in a single pass.

2.5 Floating Point Precision

The CPU implementation of the XGBoost algorithm represents gradient/Hessian pairs using two 32 bit floats. All intermediate summations are performed using 64 bit doubles to control loss of precision from floating point addition. This is problematic when using GPUs as the number of intermediate values involved in a reduction scales with the input size. Using doubles significantly increases the usage of scarce registers and shared memory; moreover, gaming GPUs are optimised for 32 bit floating point operations and give relatively poor double precision throughput.

Table 10 shows the theoretical GFLOPs of two cards we use for benchmarking. The single precision GFLOPs are calculated as 2 x number of CUDA cores x core clock speed (in GHz), where the factor of 2 represents the number of operations per required FMA (fused-multiply-add) instruction. Both these cards have 32 times more single precision ALUs (arithmetic logic units) than double precision ALUs, resulting in 1/32 the theoretical double precision performance. Therefore an algorithm relying on double precision arithmetic will have severely limited performance on these GPUs.

We test the loss of precision from 32 bit floating point operations to see if double precision is necessary. We test 32 bit parallel and sequential summation by summing over a large array of random numbers. Sequential double precision summation is used as the baseline, with the error measured as the absolute difference from the baseline. The experiment is performed over 10 million random numbers between -1 and 1, with 100 repeats. The mean error and standard deviation are reported in Table 11. The Thrust library is used for parallel GPU reduction based on single precision operations.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

The 32 bit parallel summation shows dramatically superior numerical stability copmared to the 32 bit sequential summation. This is because the error of parallel summation grows proportionally to $O(\log n)$, as compared to $O(n)$ for sequential summation (Higham, 1993). The parallel reduction algorithm from Figure 3 is commonly referred to as "pairwise summation" in literature relating to floating point precision. The average error of 0.0007 over 10 million items shown in Table 11 is more than acceptable for the purposes of gradient boosting. This also suggests that the sequential summation within the original XGBoost could be safely performed in single precision floats. A mean error of 0.0694 over 10 million items is very unlikely to be significant compared to the noise typically present in the training sets of supervised learning tasks.

2.6 Building Tree Classifiers on GPUs

GPU accelerated decision trees and forests have been studied as early as 2008 in (Sharp, 2008) for the purpose of object recognition, achieving speedups of up to 100x for this task. Decision forests were mapped to a 2-D texture array and trained/evaluated using GPU pixel and vertex shaders. A more general purpose Random Forest implementation is described in (Grahn et al., 2011) showing speedups of up to 30x over state-of-the-art CPU implementations for large numbers of trees. The authors use an approach where one GPU thread is launched to construct each tree in the ensemble.

A decision tree construction algorithm using CUDA based on 'SPRINT: A Scalable Parallel Classifier for Data Mining' is described in (Chiu et al., 2011). No performance results are reported. Another decision tree construction algorithm is described in (Lo et al., 2014). They report speedups of 5-55x over WEKA's Java based implementation of C4.5 (Quinlan, 2014), called J48, and 18x over SPRINT. Their algorithm processes one node at a time and as a result scales poorly at higher tree depths due to higher per-node overhead as compared to a CPU algorithm.

Nasridinov *et al.* describe a GPU accelerated algorithm for ID3 decision tree construction in (Nasridinov et al., 2014), showing moderate speedups over WEKA's ID3 implementation. Nodes are processed one at a time and instances are resorted at every node. Strnad *et al.* (2016) devise a decision tree construction algorithm that stores batches of nodes in a work queue on the host and processes these units of work on the GPU. They achieve speedups of between 2-7x on large data sets as compared to a multithreaded CPU implementation. Instances are resorted at every node (Strnad and Nerat, 2016).

Our work has a combination of key features differentiating it from these previous approaches. Firstly, our implementation processes all nodes in a level concurrently, allowing it to scale beyond trivial depths with near constant run time. A GPU tree construction algorithm that processes one node at a time will incur a nontrivial constant kernel launch overhead (discussed in Section 2.3.4) for each node processed. Additionally, as the training set is recursively partitioned at each level, the average number of training examples in each node decreases rapidly. Processing a small number of training examples in a single GPU kernel will severely underutilise the device. This means the run-time increases dramatically with tree depth. To achieve state-of-the-art results in data mining competitions we found that users very commonly required tree depths of greater than 10 in XGBoost. This contradicts the conventional wisdom that a tree depth of between 4-8 is sufficient for most boosting applications (Friedman et al., 2001). Our approach of processing all nodes on a level concurrently is far more practical in this setting.

Secondly, our decision tree implementation is not a hybrid CPU/GPU approach and so does not use the CPU for computation. We find that all stages of the tree construction algorithm may be efficiently completed on the GPU. This was a conscious design decision in order to reduce the bottleneck of host/device memory transfers. Host/device transfers are at the time of writing limited by the bandwidth of the Gen 3 PCIe standard to approximately 16 GB/s. The Titan X we use for benchmarking has an on device memory bandwidth of 480 GB/s, a factor of 30 times greater. Consequently, applications that move data back and forward between the host and device will not be able to achieve peak performance. Building the entire decision tree in device memory has the disadvantage that the capacity of device memory is often significantly less than host memory. Despite this, we show that it is possible to process some very large benchmark datasets entirely in device memory on a commodity GPU.

Thirdly, our algorithm implements the sparsity aware tree construction method introduced by XGBoost. This allows it to efficiently process sparse input matrices in terms of run-time and memory usage. This is in contrast to all previous GPU tree construction algorithms.

Additionally our implementation is provided as a part of a fully featured machine learning library. It implements regression, binary classification, multiclass classification and ranking through the generalised

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

	f0			f1	f2			
Node Id	0	0	0	0	0	0	0	0
Instance Id	0	2	3	3	2	0	1	3
Feature value	0.1	0.5	0.9	5.2	3.1	3.6	3.9	4.7

Table 12. Device Memory Layout: Feature Values

Instance Id	0	1	2	3
Gradient pair	p_0	p_1	p_2	p_3

Table 13. Device Memory Layout: Gradient Pairs

Thread block 0 $\Longrightarrow$

	f0							
Instance Id	0	2	3	1	7	5	6	4
Feature value	0.1	0.2	0.3	0.5	0.5	0.7	0.8	0.8
Gradient pair	p_0	p_2	p_3	p_1	p_7	p_5	p_6	p_4

Table 14. A Single Thread Block Evaluating Splits

gradient boosting framework of XGBoost and has an active user base. No published implementations exist for any of the existing GPU tree construction algorithms described above, making direct comparison to the approach presented in this work impractical.

3 PARALLEL TREE CONSTRUCTION

Our algorithm builds a single decision tree for a boosting iteration by processing decision tree nodes in a level-wise manner. At each level we search for the best splits within each leaf node, update the positions of training instances based on these new splits and then repartition data if necessary. Processing an entire level at a time allows us to saturate the GPU with the maximum amount of work available in a single iteration. Previous approaches such as (Lo et al., 2014) and (Nasridinov et al., 2014) process a single node at a time and scale poorly at higher tree depths because they sequentially execute small batch sizes. Our algorithm performs the following three high level phases for each tree level until the maximum tree depth is reached: (1) find splits, (2) update node positions, and (3) sort node buckets (if necessary).

3.1 Phase 1: Find splits

The first phase of the algorithm finds the best split for each leaf node at the current level.

3.1.1 Data Layout

To facilitate enumeration through all split points, the feature values should be kept in sorted order. Hence, we use the device memory layout shown in Tables 12 and 13. Each feature value is paired with the ID of the instance it belongs to as well as the leaf node it currently resides in. Data is stored in sparse column major format and instance IDs are used to map back to gradient pairs for each instance. All data is stored in arrays in device memory. The tree itself can be stored in a fixed length device array as it is strictly binary and has a maximum depth known ahead of time.

3.1.2 Block Level Parallelism

Given the above data layout notice that each feature resides in a contiguous block and may be processed independently. In order to calculate the best split for the root node of the tree we greedily select the best split within each feature, delegating a single thread block per feature. The best splits for each feature are then written out to global memory and are reduced by a second kernel. A downside of this approach is that when the number of features is not enough to saturate the number of streaming multiprocessors—the hardware units responsible for executing a thread block—the device will not be fully utilised.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

	f0			f1	f2			
Node Id	2	1	2	2	1	2	1	2
Instance Id	0	2	3	3	2	0	1	3
Feature value	0.1	0.5	0.9	5.2	3.1	3.6	3.9	4.7

Table 15. Interleaved Node Buckets

	f0			f1	f2			
Node Id	1	2		2	1		2	
Instance Id	0	2	3	3	2	1	0	3
Feature value	0.5	0.1	0.9	5.2	3.1	3.9	3.6	4.7

Table 16. Sorted Node Buckets

3.1.3 Calculating Splits

In order to calculate the best split for a given feature we evaluate Equation 4 at each possible split location. This depends on (G_L, H_L) and (G_R, H_R). We obtain (G_L, H_L) from a parallel scan of gradient pairs associated with each future value. (G_R, H_R) can be obtained by subtracting (G_L, H_L) from the node total which we know from the parent node.

The thread block moves from left to right across a given feature, consuming "tiles" of input. A tile here refers to the set of input items able to be processed by a thread block in one iteration. Table 14 gives an example of a thread block with four threads evaluating a tile with four items. For a given tile, gradient pairs are scanned and all splits are evaluated.

Each 32 thread warp performs a reduction to find the best local split and keeps track of the current best feature value and accompanying gradient statistics in shared memory. At the end of processing the feature another reduction is performed over all the warps' best items to find the best split for the feature.

3.1.4 Missing Values

The original XGBoost algorithm accounts for missing values by scanning through the input values twice as described in Section 2.1.6—once in the forwards direction and once in the reverse direction. An alternative method employed by our GPU algorithm is to perform a sum reduction over the entire feature before scanning. The gradient statistics for the missing values can then be calculated as the node sum statistics minus the reduction. If the sum of the gradient pairs from the missing values is known only a single scan is then required. This method was chosen as the cost of a reduction can be significantly less than performing the second scan.

3.1.5 Node Buckets

So far the algorithm description only explains how to find a split at the first level where all instances are bucketed into a single node. A decision tree algorithm must by definition separate instances into different nodes and then evaluate splits over these subsets of instances. This leaves us with two possible options for processing nodes. The first is to leave all data instances in their current location, keeping track of which node they currently reside in using an auxiliary array as shown in Table 15. When we perform a scan across all data values we keep temporary statistics for each node. We therefore scan across the array processing all instances as they are interleaved with respect to their node buckets. This is the method used by the CPU XGBoost algorithm. We also perform this method on the GPU but only to tree depths of around 5. This interleaved algorithm is fully described in Section 3.1.6.

The second option is to radix sort instances by their node buckets at each level in the tree. This second option is described fully in Section 3.1.7. Briefly, data values are first ordered by their current node and then by feature values within their node buckets as shown in Table 16. This transforms the interleaved scan ("multiscan") problem described above into a segmented scan, which has constant temporary storage requirements and thus scales to arbitrary depths in a GPU implementation.

In our implementation we use the interleaved method for trees of up to depth 5 and then switch to the sorting version of the algorithm. Avoiding the expensive radix sorting step for as long as possible can provide speed advantages, particularly when building small trees. The maximum number of leaves at depth five is 32. At greater depths there are insufficient shared memory resources and the exponentially

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Algorithm 5: Reduction - Thread block execution

1. An input tile is loaded.
2. Each warp performs local reduction for each bucket, masking off items for the current bucket.
3. Each warp adds its local reductions into shared memory.
4. The remaining tiles are processed.
5. The partial sums in shared memory are reduced by a single warp into the final node sums.

Algorithm 6: Multiscan - Thread block execution

1. An input tile is loaded.
2. Each warp performs local scans for each bucket, masking off items for the current bucket.
3. The sums from each local scan are placed into shared memory.
4. The partial sums in shared memory are scanned.
5. The scanned partial sums in shared memory are added back into the local values.
6. The running sum from the previous tile is added to the local values.
7. The remaining tiles are processed.

increasing run-time begins to be uncompetitive. We first describe the interleaved algorithm in Section 3.1.6 before discussing the algorithm based on radix sort in Section 3.1.7.

3.1.6 Interleaved Algorithm: Finding a Split

In order to correctly account for missing values a multireduce operation must be performed to obtain the sums within interleaved sequences of nodes. A multiscan is then performed over gradient pairs. Unique feature values are then identified and gain values calculated to identify the best split for each node.

Multireduce and Multiscan

Algorithms 5 and 6 outline the approach used for multireduce/multiscan at the thread block level. Our multiscan/multireduce approach is formulated around sequentially executing fast warp synchronous scan/reduce operations for each bucket. Passing vectors of items to the binary associative operator is not generally possible given the number of buckets and the limited temporary storage. This was discussed in Section 2.4. We instead perform warp level multiscan operations. Listing 7 shows how a 32 thread warp can perform a multiscan by masking off non-active node buckets and performing a normal warp scan for each node bucket. The function 'WarpExclusiveScan()' here refers to an exclusive version of the warp scan described in Listing 3.

Note that n many warp reductions/scans must be performed over every 32 items where n is the number of buckets (equal to the number of active leaves). This leads to an exponentially increasing run time relative to the depth of the tree but is surprisingly performant even up to $n = 32$ as warp synchronous reductions/scans using shuffle instructions are cheap to compute. They only perform operations on registers and incur no high latency reads or writes into global memory.

The exclusive scan for the entire input tile is calculated from individual warp scans by performing the same multiscan operation over the sums of each warp scan and scattering the results of this back into each item. More detailed information on how to calculate a block wide scan from smaller warp scan operations is given in (Nvidia, 2016).

Evaluating splits

There is one additional problem that must be solved. It arises as a consequence of processing node buckets

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Listing 7. Warp Multiscan

```
gpu_gpair gpair; //Gradient values for current item
int node_id; //Node bucket of current item
gpu_gpair exclusive_scan_output;

for (int NODE = 0; NODE < N_NODES; NODE++) {
  bool node_active = node_id == NODE;

  gpu_gpair scan_result;
  gpu_gpair node_sum;

  //First argument is the scan input
  //Result is placed in the second argument
  //Warp sum is placed in the third argument
  WarpExclusiveScan(node_active ? gpair : gpu_gpair(),
    scan_result, node_sum);

  if (node_active) {
    exclusive_scan_output = scan_result;
  }
}
```

in interleaved order. In a decision tree algorithm, when enumerating through feature values to find a split, we should not choose a split that falls between two elements with the same value. This is because a decision rule will not be able to separate elements with the same value. For a value to be considered as a split the corresponding item must be the leftmost item with that feature value for that particular node (we could also arbitrarily take the rightmost value).

Because the node buckets are interleaved it is not possible to simply check the item to the left to see if the feature value is the same—the item to the left of a given item may reside in a different node. To check if an item with a certain feature value is the leftmost item with that value in its node bucket we can formulate a scan with a special binary associative operator. First each item is assigned a bit vector $\vec{x}$ of length $n+1$ where n is the number of buckets. If the item resides within bucket i then x_i will be set to 1. If the item's feature value is distinct from the value of the item directly to the left (irrespective of bucket) then x_{n+1} is set to 1. All other bits are set to 0.

We can then define a binary associative operator as follows:

$$op(\vec{a},\vec{b}) = \begin{cases} \vec{b}, & \text{if } b_{n+1} = 1 \\ \vec{a} \vee \vec{b}, & \text{otherwise} \end{cases} \tag{5}$$

Bit x_{n+1} acts as a segmentation flag, resetting the scan so many small scans are performed across groups of items with the same feature value. Scanning the bucket flags with a logical *or* operator determines which node buckets are represented in the items to the left of the current item. Therefore within a group of items with the same feature value, if the current item's bucket flag is set to 0 for the bucket it resides in, the item represents the leftmost item with that value in its bucket. This item can then be used as a split point.

In practice a 64-bit integer is used as the bit vector in order to hold a maximum of 33 bits at the 6th level of the tree. The operator is formulated according to Listing 8 in C++ code. Moreover, when applying this interleaved algorithm we cannot choose the split value as the halfway point between two training examples: We do not know the value of the item to the left within the current node, only if it is the same as the current item or not. The split value is accordingly calculated as the current value minus some small constant. This distinction in the split value does not affect accuracy in practice.

Complete Algorithm

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Listing 8. Binary associative operator

```
BitFlagSet op(const BitFlagSet &a, const BitFlagSet &b) {
  if (check_bit(b, 63)) {
    return b;
  } else {
    return a | b;
  }
}
```

Algorithm 7: Interleaved algorithm - Thread block execution

1. Load input tile
2. Multireduce tile gradient pairs
3. Go to 1. until all tiles processed
4. Return to first tile
5. Load input tile
6. Multiscan tile gradient pairs
7. Scan tile for unique feature values
8. Calculate gain for each split
9. Store best split for each warp
10. Go to 5. until all tiles processed
11. Output best splits

Given a reduction, scan and the above method for finding unique feature values we have all the machinery necessary to enumerate splits and select the best. The complete algorithm for a thread block processing a single feature at a given tree level is shown in Algorithm 7.

The output of this algorithm contains the best splits for each leaf node for a given feature. Each thread block outputs the best splits for its assigned feature. These splits are then further reduced by a global kernel to find the best splits for any feature.

3.1.7 Sorting Algorithm: Finding a Split

The sorting implementation of the split finding algorithm operates on feature value data grouped into node buckets. Given data sorted by node ID first and then feature values second we can perform segmented scan/reduce operations over an entire feature only needing a constant amount of temporary storage.

The segmented reduction to find gradient pair sums for each node is implemented as a segmented sum scan, storing the final element from each segment as the node sum. Another segmented scan is then performed over the input feature to get the exclusive scan of gradient pairs. After scanning each tile the split gain is calculated using the scan and reduction as input and the best splits are stored in shared memory.

The segmented scan is formulated by performing an ordinary scan over key value pairs with a binary associative operator that resets the sum when the key changes. In this case the key is the current node bucket and the value is the gradient pair. The operator is shown in Equation 6.

$$op(a_{key}, a_{value}, b_{key}, b_{value}) = \begin{cases} (b_{key}, b_{value}), & \text{if } a_{key} \neq b_{key} \\ (b_{key}, a_{value} + b_{value}), & \text{otherwise} \end{cases} \quad (6)$$

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Algorithm 8: Sorting Algorithm Split Finding - Thread block execution

1. Load input tile
2. Segmented reduction over tile gradient pairs
3. Go to 1. until all tiles processed
4. Return to first tile
5. Load input tile
6. Segmented scan over tile gradient pairs
7. Calculate gain for each split
8. Store best split for each warp
9. Go to 5. until all tiles processed
10. Output best splits

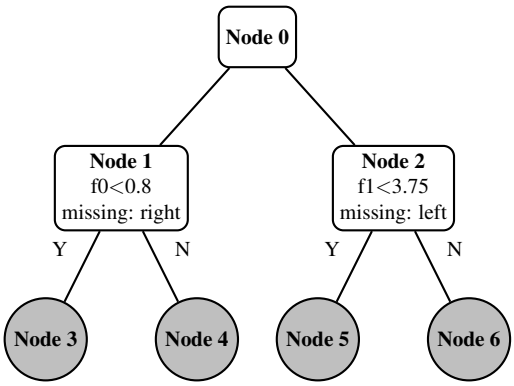

Figure 9. Decision tree: four new leaves

An overview of the split finding algorithm for a single thread block processing a feature is provided in Algorithm 8. The output of this algorithm, like that of the interleaved algorithm, consists of the best splits for each feature, and each node. This is reduced by a global kernel to find the best splits for each node, of any feature.

3.2 Phase 2: Update Node Positions

Once the best splits for each node have been calculated, the node positions for each instance must be updated. This is made non-trivial because of the presence of missing values. We first create an array containing the pre-split node position of each training instance. These node positions are then updated as if they contained all missing values, according to the default missing direction in the newly calculated splits. We then update this array again based on the feature values of the instances. Any instance which does not have a value for that feature (missing value) will have its node position left unchanged as per the missing direction. Because we now have the updated node position for each instance we write these node positions back to each feature value.

To illustrate this with an example, Figure 9 shows the state of a decision tree after having calculated splits for level 1. The node positions in the data structure used for split finding (Table 17) must be updated before proceeding to calculate the splits for level 2. To do this we update the array in Table 18 that maps instances to a node.

First we update the node ID map in the missing direction. All instances residing in node 1 are updated in the right direction to node 4. Instances residing in node 2 are updated in the left direction to node 5.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

	f0			f1			
Node Id	1	2	2	1	1	2	2
Instance Id	0	2	1	3	0	1	2
Feature value	0.75	0.5	0.9	2.7	4.1	3.6	3.9

Table 17. Per Feature Value Array

Instance Id	0	1	2	3
Node Id	1	2	2	1

Table 18. Node ID map

Instance Id	0	1	2	3
Node Id	4	5	5	4

Table 19. Updated missing direction

Instance Id	0	1	2	3
Node Id	3	5	6	4

	f0			f1			
Node Id	1	2	2	1	1	2	2
Instance Id	0	2	1	3	1	1	2
Feature value	0.75	0.5	0.9	2.7	4.1	3.6	3.9

Table 20. Node ID map: Update based on feature value

	f0			f1			
Node Id	3	6	5	4	3	5	6
Instance Id	0	2	1	3	0	1	2
Feature value	0.75	0.5	0.9	2.7	4.1	3.6	3.9

Table 21. Per Feature Value Array: Updated

The node ID map now looks like Table 19.

We now update the map again using the feature values from Table 17, overwriting the previous values. Instance 0 resides in node 1 so we check if $f0 < 0.8$. This is true so instance 0 moves down the left branch into node 3. Instance 1 moves into node 5 and instance 2 moves into node 6 based on their f1 values. Note that instance 3 has a missing value for f0. Its node position is therefore kept as the missing direction updated in the previous step. This process is shown in Figure 20.

The per instance node ID array is now up-to-date for the new level so we write these values back into the per feature value array, giving Table 21.

3.3 Phase 3: Sort Node Buckets

If the sorting version of the algorithm is being used, the feature values need to be sorted by node position. If the interleaved version of the algorithm is being used (for example, in early tree levels) this step is unnecessary. Each feature value with its updated node position is sorted such that each node bucket resides in contiguous memory. This is achieved using a segmented key/value radix sort. Each feature represents a segment, the sort key is the node position and the feature value/instance ID tuple is the value. We us the segmented radix sort function from the CUB library. It delegates the sorting of each feature segment to a separate thread block. Note that radix sorting is stable so the original sorted order of the feature values will be preserved within contiguous node buckets, after sorting with node position as the key.

4 EVALUATION

The performance and accuracy of the GPU tree construction algorithm for XGBoost is evaluated on several large datasets and two different hardware configurations and also compared to CPU based XGBoost on a 24 core Intel processor.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Configuration	CPU	GHz	Cores	CPU arch.
#1	Intel i5-4590	3.30	4	Haswell
#2	Intel i7-6700K	4.00	4	Skylake
#3	2x Intel Xeon E5-2695 v2	2.40	24	Ivy Bridge

Configuration	GPU	GPU memory (GB)	GPU arch.
#1	GTX 970	4	Maxwell
#2	Titan X	12	Pascal
#3	-	-	-

Table 22. Hardware Configurations

Dataset	Training Instances	Test Instances	Features
YLTR[a]	473,134	165,660	700
Higgs[b]	10,500,000	500,000	28
Bosch[c]	1,065,373	118,374	968

Table 23. Datasets

[a] https://webscope.sandbox.yahoo.com/catalog.php?datatype=c
[b] https://archive.ics.uci.edu/ml/datasets/HIGGS
[c] https://www.kaggle.com/c/bosch-production-line-performance/data

Dataset	objective	eval_metric	max_depth	eta	boosting iterations
YLTR	rank:ndcg	ndcg@10	6	0.1	500
Higgs	binary:logistic	auc	12	0.1	500
Bosch	binary:logistic	auc	6	0.1	500

Table 24. Parameters

Hardware configurations are described in Table 22. On configuration #1, where there is limited device memory, a subset of rows from each dataset is taken in order to fit within device memory.

Datasets are described in Table 23 and parameters used for each dataset are shown in Table 24.

For the YLTR dataset we use the supplied training/test split. For the Higgs dataset we randomly select 500,000 instances for the test set, as in (Chen and Guestrin, 2016). For the Bosch dataset we randomly sample 10% of the instances for the test set and use the rest for the training set.

We use 500 boosting iterations for all datasets unless otherwise specified. This is a common real world setting that provides sufficiently long runtimes for benchmarking. We set η (the learning rate) to 0.1 as the XGBoost default of 0.3 is too high for the number of boosting iterations. For the YLTR and Bosch datasets we use the default tree depth of six because both of these datasets tend to generate small trees. The Higgs dataset results in larger trees so we can set max depth to 12, allowing us to test performance for large trees. Both the Higgs and Bosch datasets are binary classification problems so we use the binary:logistic objective function for XGBoost. Both Higgs and Bosch also exhibit highly imbalanced class distribution, so the AUC (area under curve) evaluation metric is appropriate. For the YLTR dataset we use the rank:ndcg objective and ndcg@10 evaluation metric to be consistent with the evaluation from (Chen and Guestrin, 2016). All other XGBoost parameters are left as the default values.

4.1 Accuracy

In Table 25 we show the accuracy of the GPU algorithm compared to the CPU version. We test on configuration #1 so use a subset of the training set to fit the data within device memory but use the full test set for accuracy evaluation.

There is only minor variation in accuracy between the two algorithms. Both algorithms are equivalent for the Higgs dataset, the CPU algorithm is marginally more accurate for the YLTR dataset and the GPU algorithm is marginally more accurate on the Bosch dataset. In Table 26 we also show the accuracy without using the interleaved version of the GPU algorithm. Variations in accuracy are attributable to

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Dataset	Subset	Metric	CPU accuracy	GPU accuracy
YLTR	0.75	ndcg@10	0.7784	0.7768
Higgs	0.25	auc	0.8426	0.8426
Bosch	0.35	auc	0.6833	0.6905

Table 25. Accuracy Benchmarks

Dataset	Subset	Metric	GPU accuracy (sorting version only)
YLTR	0.75	ndcg@10	0.7776
Higgs	0.25	auc	0.8428
Bosch	0.35	auc	0.6849

Table 26. Accuracy Benchmarks - Sorting version only

Dataset	Subset	CPU time (s)	GPU time (s)	Speedup
YLTR	0.75	1577	376	4.19
Higgs	0.25	7961	1201	6.62
Bosch	0.35	1019	249	4.09

Table 27. Configuration #1 Speed Benchmarks

Dataset	Subset	CPU time (s)	GPU time (s)	Speedup
YLTR	1.0	877	277	3.16
Higgs	1.0	14504	3052	4.75
Bosch	1.0	3294	591	5.57

Table 28. Configuration #2 Speed Benchmarks

the interleaved version of the algorithm not choosing splits at the halfway point between two training examples, instead choosing the split value as the right most training example minus some constant. Differences also occur due to floating point precision as discussed in Section 2.5.

4.2 Speed

Tables 27 and 28 show the relative speed up of the GPU algorithm compared to the CPU algorithm over 500 boosting iterations. For configuration #1 with lower end desktop hardware, speed ups of between 4.09x and 6.62x are achieved. On configuration #2 with higher end desktop hardware but the same number of cores, speed ups of between 3.16x an 5.57x are achieved. The GTX 970 used in configuration #1 must sample the datasets as they do not fit entirely in device memory. The Titan X used in configuration #2 is able to fit all three datasets entirely into memory.

Figure 10 shows the performance of the GPU algorithm across varying problem sizes using configuration #1. The experiment is performed on subsets of the Bosch dataset using 20 boosting iterations. The GPU algorithm's time increases linearly with respect to the number of input rows. It is approximately equal to the CPU algorithm at 10,000 rows and always faster thereafter for this dataset. This gives an idea of the minimum batch size at which the GPU algorithm begins to be effective.

In Figure 11 we show the performance of the Titan X from configuration #2 against configuration #3 (a high-end 24 core server) on the Yahoo dataset with 500 boosting iterations and varying numbers of threads. The Titan X outperforms the 24 core machine by approximately 1.2x, even if the number of threads for the 24 core machine is chosen optimally.

4.3 Interleaved algorithm performance

In Table 29 and Figure 12 we show the effect of changing the threshold at which the algorithm switches between the interleaved version of the algorithm and the sorting version of the algorithm. Timings are from 100 boosting iterations on a 35% subset of the Bosch dataset using configuration #1. Using the interleaved version of the algorithm shows benefits all the way up to the fifth level with a 1.14x speed increase as compared to just using the sorting algorithm. After this depth temporary storage is insufficient

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

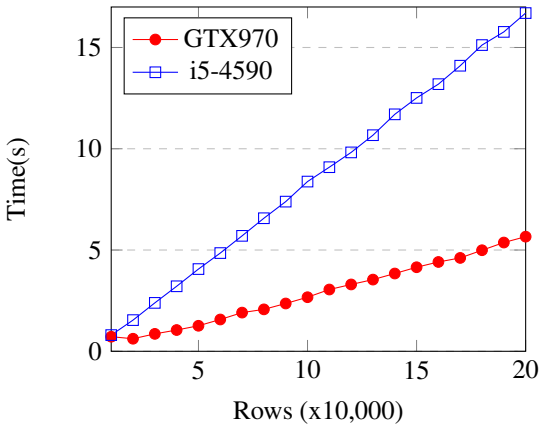

Figure 10. Bosch: Time vs Problem Size

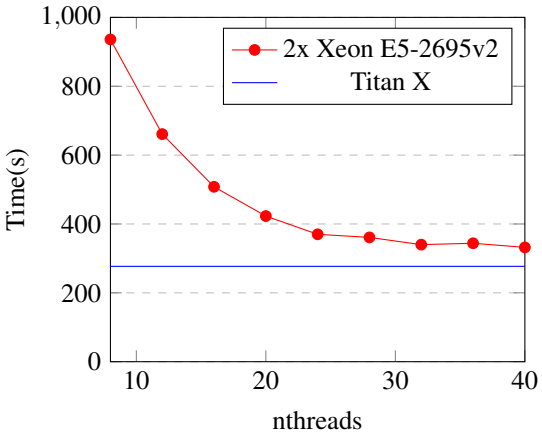

Figure 11. Yahoo LTR: nthreads vs time

Levels	GPU time (s)	Accuracy	Speedup
0	85.96	0.7045	1.0
1	85.59	0.7102	1.0
2	82.32	0.7047	1.04
3	79.97	0.7066	1.07
4	76.38	0.7094	1.13
5	75.21	0.7046	1.14

Table 29. Bosch dataset: Interleaved levels

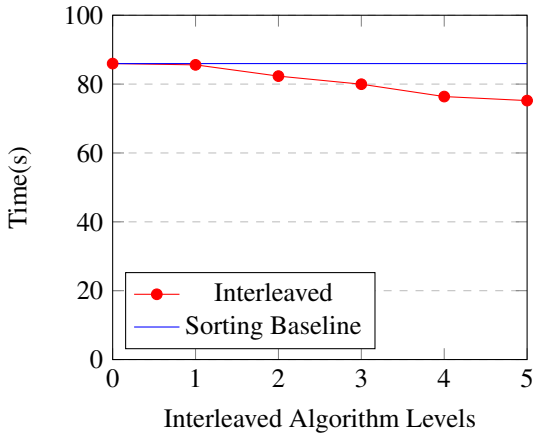

Figure 12. Bosch: Interleaved algorithm threshold

to keep using the interleaved approach. Note that for the first level the interleaved algorithm and the sorting algorithm are equivalent as there is only one node bucket.

The accuracy shows some variance as the interleaved split finding algorithm records feature splits in a slightly different way as compared to the sorting algorithm. Both versions split the training set in exactly the same place but the sorting version records the feature value for the split as halfway between two instances and the interleaved version records the split point as slightly less than the rightmost instance. Because of this, when we use the model on the unseen test set the results can be marginally different for the two versions.

Surprisingly the interleaved algorithm is still faster than the sorting algorithm at level 5 despite the fact that the multiscan and multireduce operations must sequentially iterate over $2^5 = 32$ nodes at each step. This shows that executing instructions on elements held in registers or shared memory carries a very low cost relative to uncoalesced reordering of elements in device memory, as is performed when radix sorting.

4.4 Memory consumption

We show the device memory consumption in Table 30 for all three benchmark datasets. Each dataset can be fit entirely within the 12GB device memory of a Titan X card.

In Table 31 we show the memory consumption of the original CPU algorithm for comparison. Host memory consumption was evaluated using the valgrind massif[2] heap profiler tool. Device memory usage

[2]http://valgrind.org/docs/manual/ms-manual.html

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Dataset	Device memory (GB)
YLTR	4.03
Higgs	11.32
Bosch	8.28

Table 30. Memory: GPU Algorithm

Dataset	Host memory (GB)
YLTR	1.80
Higgs	6.55
Bosch	3.28

Table 31. Memory: CPU Algorithm

was recorded programmatically using custom memory allocators. The device memory requirements are approximately twice that of the original CPU algorithm. This is because the CPU algorithm is able to process data in place, whereas the GPU algorithm requires sorting functions that are not in place and must maintain separate buffers for input and output.

5 CONCLUSION

A highly practical GPU accelerated tree construction algorithm is devised and evaluated within the XGBoost library. The algorithm is built on top of efficient parallel primitives and switches between two modes of operation depending on tree depth. The ”interleaved” mode of operation shows that multiscan and multireduce operations with a limited number of buckets can be used to avoid expensive sorting operations at tree depths below six, resulting in speed increases of 1.14x for the GPU implementation.

The GPU algorithm provides speedups of between 3-6x over multicore CPUs on desktop machines and a speed up of 1.2x over 2x Xeon CPUs with 24 cores. We see significant speedups for all parameters and datasets above a certain size, while providing an algorithm that is feature complete and able to handle sparse data. Potential drawbacks of the algorithm are that the entire input matrix must fit in device memory and device memory consumption is approximately twice that of the host memory used by the CPU algorithm. Despite this, we show that the algorithm is memory efficient enough to process the entire Higgs dataset containing 10 million instances and 28 features on a single 12GB card.

Our algorithm provides a practical means for XGBoost users processing large data sets to significantly reduce processing times, showing that gradient boosting tasks are a good candidate for GPU acceleration and are therefore no longer solely the domain of multicore CPUs.

REFERENCES

Baxter, S. (2013). Modern GPU - performance. `https://nvlabs.github.io/moderngpu/performance.html`. Accessed: 2016-06-14.

Blelloch, G. E. (1990). Prefix sums and their applications. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University.

Boyer, M. (2016). CUDA kernel overhead. `https://www.cs.virginia.edu/~mwb7w/cuda_support/kernel_overhead.html`. Accessed: 2016-11-21.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794. ACM.

Chiu, C.-C., Luo, G.-H., and Yuan, S.-M. (2011). A decision tree using CUDA GPUs. In *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, pages 399–402. ACM.

Coates, A., Huval, B., Wang, T., Wu, D., Catanzaro, B., and Andrew, N. (2013). Deep learning with COTS HPC systems. In *Proceedings of The 30th International Conference on Machine Learning*, pages 1337–1345.

Eilers, M. (2014). Multireduce and multiscan on modern GPUs. Master’s thesis, Department of Computer Science, University of Copenhagen.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin.

Grahn, H., Lavesson, N., Lapajne, M. H., and Slat, D. (2011). CudaRF: A CUDA-based implementation of random forests. In *Proceedings of the 9th IEEE/ACS International Conference on Computer Systems and Applications*, pages 95–101.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18.

Harris, M. (2007). Optimizing parallel reduction in CUDA. `http://developer.download.nvidia.com/assets/cuda/files/reduction.pdf`. Accessed: 2017-03-31.

Harris, M. (2013). How to access global memory efficiently in CUDA C/C++ kernels. `https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/`. Accessed: 2016-11-24.

Harris, M., Sengupta, S., and Owens, J. D. (2007). Parallel prefix sum (scan) with CUDA. *GPU gems*, 3(39):851–876.

Higham, N. J. (1993). The accuracy of floating point summation. *SIAM Journal on Scientific Computing*, 14(4):783–799.

Hillis, W. D. and Steele Jr, G. L. (1986). Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183.

Hoberock, J. and Bell, N. (2017). Thrust: A parallel template library. `https://thrust.github.io/`.

Lo, W.-T., Chang, Y.-S., Sheu, R.-K., Chiu, C.-C., and Yuan, S.-M. (2014). CUDT: a CUDA based decision tree algorithm. *The Scientific World Journal*, 2014.

Matloff, N. (2011). *Programming on parallel machines*. `http://heather.cs.ucdavis.edu/~matloff/158/PLN/ParProcBook.pdf`.

Merrill, D. and Grimshaw, A. (2011). High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(02):245–272.

Merrill, D. and NVIDIA-Labs (2016). CUDA UnBound (CUB) Library. `https://nvlabs.github.io/cub/`.

Nasridinov, A., Lee, Y., and Park, Y.-H. (2014). Decision tree construction on GPU: ubiquitous parallel computing approach. *Computing*, 96(5):403–413.

Nvidia (2016). Block scan algorithms. `https://nvlabs.github.io/cub/namespacecub.html#abec44bba36037c547e7e84906d0d23ab`. Accessed: 2016-12-30.

Nvidia (2017). CUDA C programming guide. `http://docs.nvidia.com/cuda/index.html`.

Quinlan, J. R. (2014). *C4.5: Programs for machine learning*. Elsevier.

Sharp, T. (2008). Implementing decision trees and forests on a GPU. In *Proceedings of the 10th European Conference on Computer Vision*, pages 595–608. Springer.

Strnad, D. and Nerat, A. (2016). Parallel construction of classification trees on a GPU. *Concurrency and Computation: Practice and Experience*, 28(5):1417–1436.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.2911v1 | CC BY 4.0 Open Access | rec: 4 Apr 2017, publ: 4 Apr 2017