CS359 Computer Architecture Project Report Part B: GPU or Multi-core Programming Project

Weichen Li
5120309662
Department of Computer Science and Engineering
School of Electronic Information
and Electrical Engineering
Shanghai Jiaotong University
Shanghai China 200240
Email: eizo.lee@sjtu.edu.cn

***Abstract*—In this project we will learn to use a GPU programming tool—CUDA—to help us solving a machine learning problem on house price prediction. The process of the machine learning is based on training data and output a model which is called ”hypothesis”, then we use this model to predict house price of the given data.**

I. Introduction

Machine learning algorithms allow computers recognize complex patterns. It focuses on the prediction, based on known properties learned from the training data. In 1959 Arthur Samuel described Machine learning as: Field of study that gives computers the ability to learn without being explicitly programmed. It has been a while machine learning was first introduced, and it is gaining popularity again with the rise of Big Data.

In this project we will introduce a way to exploit the parallelism in machine learning (because of its large scale of computation), then implement it on GPGPU—General Purpose GPU. Thus the process of learning can be dramatically faster than implemented on a single CPU and programmed with serial codes. Finally, we can derive an application called *TestLRApp.exe*, then use it to load data, train data and predict.

A. Machine Learning

Figure 1 shows how some of the machine learning processes work. On phase 1, given a data set a machine learning algorithm can recognize complex patterns and come up with a model. In most cases this phase is the bulk of the computation. In the second phase any given data can run through the model to make a prediction. For example if you have a data set of house prices by size, you could let the machine learn from the data set and let it predict house price of any given size.

It does this by recognizing the function which defines the relation between the different features of the problem. A linear problem with two dimensions, like house price (the house size is the feature and the house price is the label data), can be expressed with the $f(x) = ax + b$ model. Figure 2 shows how one feature can be used on a linear regression problem to predict new house prices. The term ”hypothesis” was used in the Stanford course to describe the model.

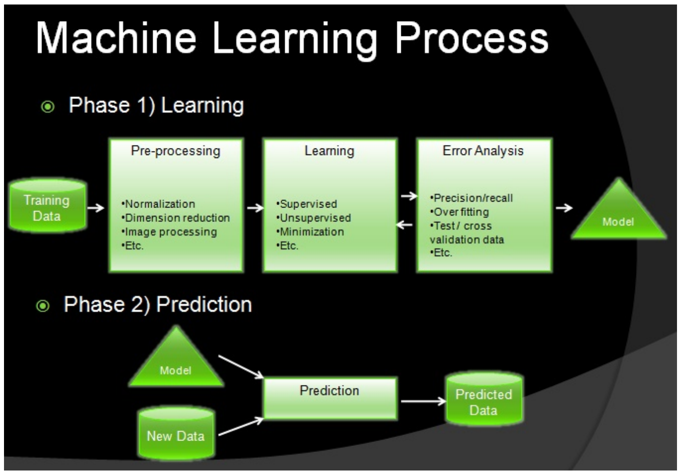


Fig. 1. Machine Learning Process

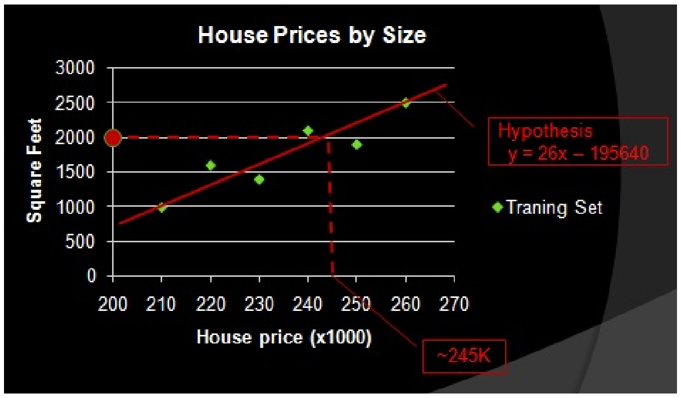


Fig. 2. Hypothesis

Depending to the data set, more complex functions can be used. On Figure 3 you can see how the complexity can grow easily from 2 dimensions linear to hundreds of dimensions polynomial. In a spam filtering problem the different features could be words in the email or in a face recognition problem the features could be the pixels of the image. In the house price prediction example, features are properties of the house

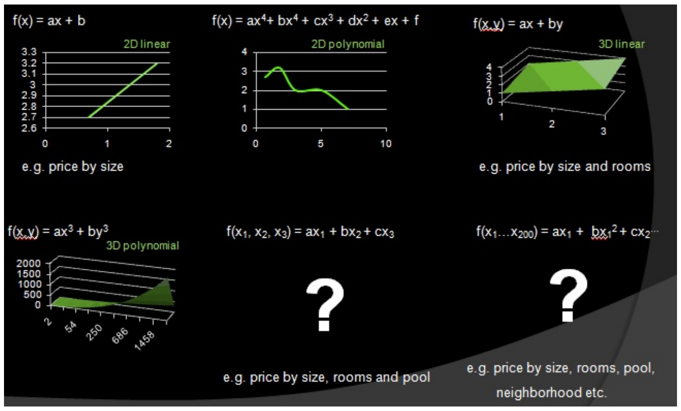


Fig. 3. Different hypothesis

which are affecting the price. e.g. size, room count, floors, neighborhood, crime rate etc.

There are many machine learning algorithms for different problem types. The most common groups of these algorithms are *Supervised Learning*, *Unsupervised Learning*. *Supervised learning* is used on problems where we can provide the output values for the learning algorithm. For example: house prices for some house features is the output value, therefore house price prediction is a supervised learning problem. Data with these output values is named as ”labeled data”. On the other hand *unsupervised learning* does not require output values, patterns or hidden structures can be recognized just with feature data. For example: clustering social data to determine groups of people by interest would not require to define any output value, therefore it is an unsupervised learning problem.

B. Gradient Descent

In supervised learning problems, the machine can learn the model and come up with a hypothesis by running a hypothesis with different variables and testing if the result is close to the provided labels (calculating the error). Figure 4 shows how a training data is plot and the error is calculated. An optimization algorithm named *Gradient Descent* (Equation 1) can be used to find the optimum hypothesis. (α is the learning rate)

In this simple two dimensional problem, the algorithm would run for every different value of ”a” and ”b”, and would try to find the minimum total error.

$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \tag{1}$$

C. Large Scale Machine Learning

Machine learning problems become computationally expensive when the complexity (dimensions and polynomial degree) increases and/or when the amount of data increases. Especially on big data sources with hundreds of millions of samples, the time to run optimization algorithms increases dramaticaly. That’s why we are looking for parallelization opportunities in the algorithms. The error summation of gradient descent algorithm is a perfect candidate for parallelization. We could split the data into multiple parts and run gradient descent on these parts in parallel. In Figure 5 you can see how the data is split into four parts and fed into four different processors. On the next step the result is gathered together to run the rest of the algorithm.

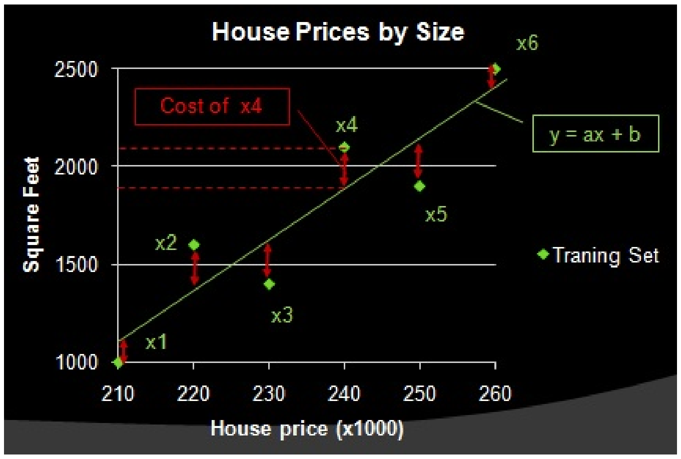


Fig. 4. Gradient Descent

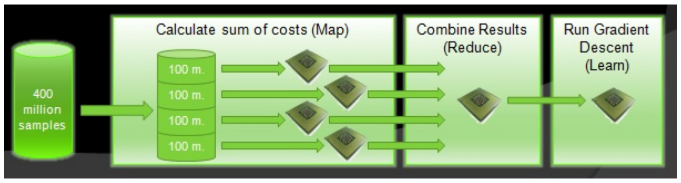


Fig. 5. Data split

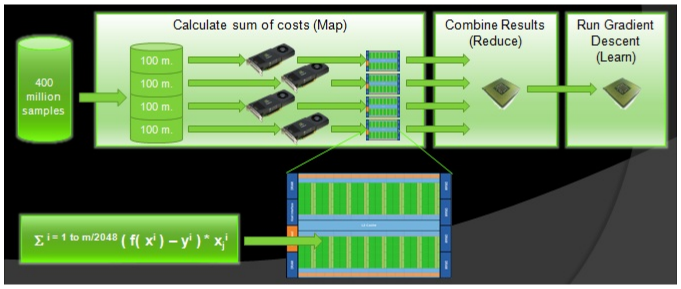

400 million samples
Calculate sum of costs (Map)
100 m.
100 m.
100 m.
100 m.
Combine Results (Reduce)
Run Gradient Descent (Learn)
Σ i = 1 to m/2048 (f(x^i) – y^i) * x_j^i

Fig. 6. Multicore configuration

Clearly, this approach would speed up the machine learning computation by almost four times. But what if we would have more cores and split the data further? That is where GPUs step into the solution. With GPUs we can parallelize in two layers: multiple GPUs and multiple cores in every GPU. Assuming a configuration with 4 GPUs and 512 cores each, we could split down the data into 512 more pieces. Figure 6 shows this configuration along with the parallelized part on the GPU cores.

D. GPGPU

Utilizing GPUs to enable dramatic increases in computing performance of general purpose scientific and engineering computing is named GPGPU. NVIDIA is providing a parallel computing platform and programming model named *CUDA* to develop GPGPU software on C, C++ or Fortran which can run on any NVIDIA GPU. NVIDIA CUDA comes with many high level APIs and libraries like basic linear algebra, FFT, imaging etc. to allow you concentrate on the business logic rather than re-writing well known algorithms.

II. Description

A. House Price Prediction Example

On this post I'll show you how you can implement house price prediction on NVIDIA CUDA. Given a house price data set based on bedrooms, square feet and year built, it is possible to let the machine learn from this data set and provide us with a model for future predictions. Because the error calculation part of the Gradient Descent algorithm is highly parallelizable, we can offload it to the GPUs.

The machine learning algorithm in this example is Polynomial Regression, a form of the well known Linear Regression algorithm. In Polynomial Regression the model is fit on a high order polynomial function. In our case we will be using bedrooms, square feet, year built, square root of bedrooms, square root of square feet, square root of year built and the product of bedrooms and square feet. The reason we added the four polynomial terms to the function is because of the nature of our data. Fitting the curve correctly is the main idea behind building a model for our machine leaning problem. Logically house prices increase by these features not in a linear or exponential way and they don't drop after a certain peek. Therefore the graph is more like a square root function, where house prices increase less and less compared to increasing any other feature.

Finding the right polynomial terms is very important for the success of the machine learning algorithm: having a very complex, tightly fitting function would generate too specific model and end up with overfitting, having a very simple function, like a straight line would generate too general model and end up with under fitting. Therefore we are using additional methods like adding regularization terms to provide a better fit to your data.

Equation 2 shows the gradient descent algorithm including with the regularization term λ.

$$\theta_j := \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \quad (2)$$

B. Application Architecture

The sample application consist of a C++ native DLL named LR_GPULib, for the machine learning implementation on the GPU and a C# Windows application named TestLRApp for the user interface. The DLL implements Data Normalization and Polynomial Regression using the high level parallel algorithm library Thrust on NVIDIA CUDA. I've mentioned on my previous blog posts about Thrust more in detail, therefore I'm not going into much detail on this post. Figure 7 shows the application architecture and also the program flow from loading the training data all the way down to making a prediction.

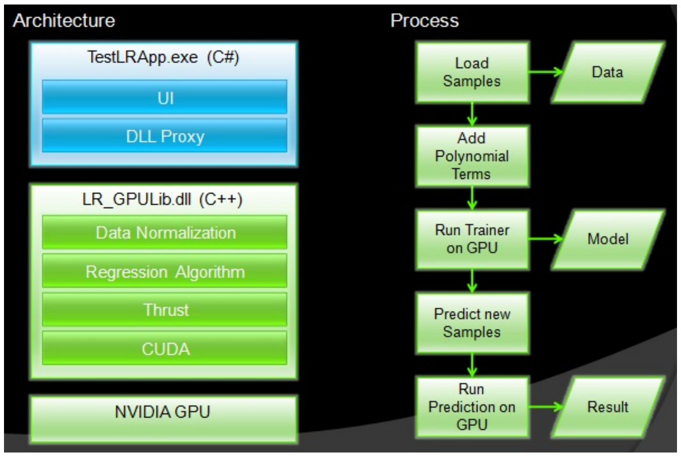
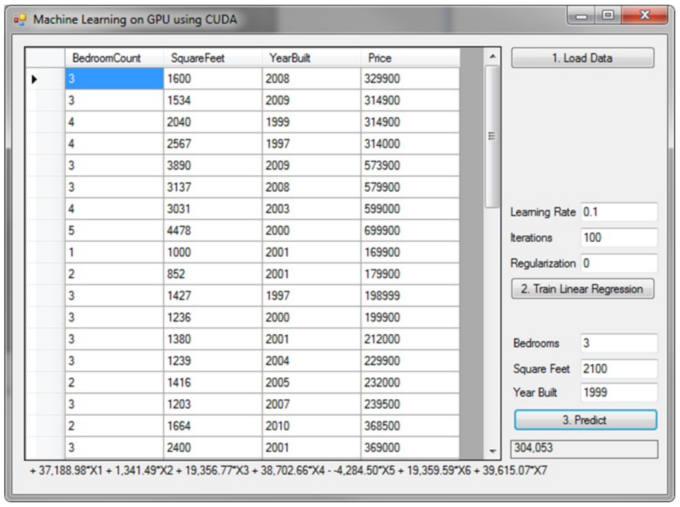


Fig. 7. Application architecture



Machine Learning on GPU using CUDA
1. Load Data
2. Train Linear Regression
3. Predict

	BedroomCount	SquareFeet	YearBuilt	Price
▶	3	1600	2008	329900
	3	1534	2009	314900
	4	2040	1999	314900
	4	2567	1997	314000
	3	3890	2009	573900
	3	3137	2008	579900
	4	3031	2003	599000
	5	4478	2000	699900
	1	1000	2001	169900
	2	852	2001	179900
	3	1427	1997	198999
	3	1236	2000	199900
	3	1380	2001	212000
	3	1239	2004	229900
	2	1416	2005	232000
	3	1203	2007	239500
	2	1664	2010	368500
	3	2400	2001	369000

Learning Rate 0.1
Iterations 100
Regularization 0

Bedrooms 3
Square Feet 2100
Year Built 1999

304,053

+ 37,188.98*X1 + 1,341.49*X2 + 19,356.77*X3 + 38,702.66*X4 - -4,284.50*X5 + 19,359.59*X6 + 39,615.07*X7

Fig. 8. User interface

The application provides the UI shown on Figure 8 to load the data, train and make a prediction with new data set. The UI also shows the hypothesis on the bottom of the dialog with all constants and features.

C. Implementation

The LR_GPU_Functors.cu file in the DLL contains the functors used as kernels on Thrust methods.

The LR_GPU.cu file in the DLL contains the normalization, learning and prediction methods.

The *Learn* method accepts the training data and the label data, which are all the features and all prices in two float arrays. The first thing the *Learn* method does is to allocate memory, add bias term and normalize the features. The reason we added the bias term is to simplify the gradient loop and the reason we normalize the features is because the data ranges are too different. E.g. square feet is four digits and bedrooms is single digit. By normalizing the features we bring them into the same range, between zero and one. Normalization gets also executed on the GPU using the *NormalizeFeatures*.

But the normalization requires the *mean* and *standard deviation (std)*, therefore mean and std are calculated first and provided to the *NormalizeFeaturesByMeanAndStd* method to calculate the mean normalization.

```
void NormalizeFeaturesByMeanAndStd(
  unsigned int trainingDataCount,
  float * d_trainingData,
  thrust::device_vector<float> dv_mean,
  thrust::device_vector<float> dv_std)
{
//Calculate mean norm: (x - mean) / std
unsigned int featureCount = dv_mean.size();
float * dvp_Mean =
   thrust::raw_pointer_cast( &dv_mean[0] );
float * dvp_Std =
   thrust::raw_pointer_cast( &dv_std[0] );
FeatureNormalizationgFunctor
  featureNormalizationgFunctor(
    dvp_Mean, dvp_Std, featureCount);

thrust::device_ptr<float>
  dvp_trainingData(d_trainingData);

thrust::transform(
  thrust::counting_iterator<int>(0),
    thrust::counting_iterator<int>
    (trainingDataCount * featureCount),
    dvp_trainingData, dvp_trainingData,
    featureNormalizationgFunctor);
}
```

The Normalization code running on the GPU is implemented in the *FeatureNormalizationgFunctor* functor, which is simply calculating $(data - mean)/std$ in parallel for every element of the data, as seen below:

```
...
__host__ __device__
float operator()(int tid,
        float trainingData)
{
//please complete the code!
//tid: [0 , trainingDataCount
              * featureCount)

int index = tid % featureCount;
if (index != 0)
  return (trainingData - meanValue[index])
                        / stdValue[index];
else
  return trainingData;

}
...
```

$$\boxed{\theta_j := \theta_j(1-\alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}}\boxed{\sum_{i=1}^{m}\boxed{(h_\theta(x^{(i)})-y^{(i)})}x_j^{(i)}}$$

Fig. 9. User interface

On the next step in the *Learn* method, the gradient descent is calculated with the for(int i = 0; i<gdIterationCount; i++) loop. As I mentioned before, the error calculation part of the gradient descent is executed in parallel but the rest is calculated sequentialy. The *thrust::transform* is used with the *TrainFunctor* to calculate *f(x)-y* in parallel for every sample. *f(x)* is simply the *A*x1 + Bx2 + Cx3 + Dx4 + Ex5 + Fx6 + Gx7 + H* hypothesis where x1 through x7 are the features (x1=bedrooms, x2=square feet, x3=year built, x4=square root of bedrooms, x5=square root of square feet, x6=square root of year built and x7=the product of bedrooms and square feet) and A through H are the constants which gradient descent will find out. This is shown with the Green Square on Figure 9. The *TrainFunctor* code snippet and the usage code snippet are shown below:

```
...
__host__ __device__
float operator()(int tid, float labelData)
{
//Please complete the code!
//tid: [0 , trainingDataCount)

float sum = 0;
for (int i = 0; i < featureCount; ++i){
  sum += hypothesis[i] *
     trainingData[featureCount * tid + i];
}
return sum - labelData;
}
...

...
thrust::transform(
      thrust::counting_iterator<int>(0),
thrust::counting_iterator<int>(
                     trainingDataCount),
dv_labelData.begin(),
               dv_costData.begin(), tf);
...
```

The *thrust::transform_reduce* is used with the *TrainFunctor2* to apply the features to the error result and sum up all of them. This is shown with the code snippet below and the Red Square on Figure 9. Rest of the *Learn* method calculates gradient descent part marked with Blue Square on Figure 9.

```
float totalCost = thrust::transform_reduce(
           thrust::counting_iterator<int>(0),
  thrust::counting_iterator<int>(
         trainingDataCount), tf2, 0.0f,
                thrust::plus<float>());
```

Once gradient descent converges, the constants A though H of the hypothesis is returned back to the TestLRApp with the *result* array.

As you may guess the prediction works by using the constants with new sample data on the hypothesis. This is done using the *Predict* method in the LR_GPULib library. As seen below the *Predict* method normalizes the given features set and calculates the hypothesis using the constants and the normalized data with the help of the *PredictFunctor*. The result is a the predicted house price for the given features.

```
...
NormalizeFeaturesByMeanAndStd(
   testDataCount, pdv_testData,
              dv_mean, dv_std);

//Predict
PredictFunctor predictFunctor(
   pdv_testData, pdv_hypothesis,
                  featureCount);
thrust::transform(
   thrust::counting_iterator(0),
thrust::counting_iterator(
  testDataCount), dv_result.begin(),
                  predictFunctor);
...

struct PredictFunctor : public
          thrust::unary_function
{
float * testData;
float * hypothesis;
unsigned int featureCount;

PredictFunctor(float * _testData,
    float * _hypothesis, unsigned
               int _featureCount)
         : testData(_testData),
       hypothesis(_hypothesis),
     featureCount(_featureCount)
{}
__host__ __device__
float operator()(int tid)
{
// Please complete the code!
int sum = 0;
sum += hypothesis[0];
for (int i = 1; i < featureCount; ++i){
sum += hypothesis[i] * testData[tid
                   * featureCount + i];
}
return sum;
}
};
```

III. Evaluation & Comparison

A. Code Addition

In this project, we have three parts of code to append. One is in *TrainFunctor*, one is in *FeatureNormalizationgFunctor* and one is in *PredictFunctor*.

1) TrainFunctor: In *TrainFunctor*, we need to calculate *f(x)-y* in parallel for every sample. *f(x)* is simply the *A*x1 + Bx2 + Cx3 + Dx4 + Ex5 + Fx6 + Gx7 + H* hypothesis where x1 through x7 are the features (x1=bedrooms, x2=square feet, x3=year built, x4=square root of bedrooms, x5=square root of square feet, x6=square root of year built and x7=the product of bedrooms and square feet) and A through H are the constants which gradient descent will find out.

So I added the code below:

```
__host__ __device__
float operator()(int tid, float labelData)
{
//Please complete the code!
//tid: [0 , trainingDataCount)

float sum = 0;
for (int i = 0; i < featureCount; ++i){
sum += hypothesis[i] *
trainingData[featureCount * tid + i];
}
return sum - labelData;
}
```

Every *tid* stands for a sample data, we can regard it as a thread. The number of threads is exactly the number of the samples.

For each thread, we will calculate the *sum* and return *sum - labelData* for that sample: in a for loop with 8 times (*featureCount* equals to 8), we add the product of a constant(*hypothesis[i]*) and a feature(*trainingData[x]*) to *sum*.

2) FeatureNormalizationgFunctor: In *FeatureNormalizationgFunctor*, which is simply calculating $(data - mean)/std$ in parallel for every element of the data, I added the code below:

```
__host__ __device__
float operator()(int tid,
float trainingData)
{
//please complete the code!
//tid: [0 , trainingDataCount
* featureCount)

int index = tid % featureCount;
if (index != 0)
return (trainingData - meanValue[index])
/ stdValue[index];
else
return trainingData;

}
```

Assume *x* is the number of training data, then we will have $8 \times x$ threads, because *featureCount* is 8, according to the source code configuration.

Thus we have to use module operation to locate the feature each thread is responsible for. However, we just perform the calculation on feature 1 to feature 7, not feature 0. Feature 0 will be used to multiply with the constant item (*H*) in training, so its value must be 1, which is already calculated in *mean* method.

3) PredictFunctor: PredictFunctor calculates the result, which is a the predicted house price for the given features, using the constants and the normalized data. I added the code below:

```
__host__ __device__
float operator()(int tid)
{
// Please complete the code!
int sum = 0;
sum += hypothesis[0];
for (int i = 1; i < featureCount; ++i){
sum += hypothesis[i] * testData[tid
* featureCount + i];
}
return sum;
}
};
```

The item *hypothesis[0]* is the constant item (*H*), so I did nothing but sorely add it to *sum*. For other hypohtesis items, each of them will multiple with the corresponding feature of the test data, then the product will be added to *sum*.

B. Results Obtaining and Evaluation

After added the missing code. I also modified some part of the source code: in file *MainForm.cs*, line 95.

1) Source Code Modification:

```
//Display the model
StringBuilder model = new StringBuilder();

model.Append(string.Format(
                 "{0:N2}",hypothesis[0]));

//featureCount == 7
for (int i = 1; i < featureCount + 1; i++)
model.Append(string.Format("{0}{1:N2}*X{2}"
          ,hypothesis[i] < 0 ? "" : "+",
                    hypothesis[i], i));
labelHypothesis.Text = model.ToString();

btn_predict.Enabled = true;
```

As you can see, I added such a line:
model.Append(string.Format(”0:N2”,hypothesis[0]));

The original version of the application didn’t output the constant item of the hypothesis, so I added it, which was *hypothesis[0]*.

TABLE I
TEST SAMPLES

Number	square feet	bedrooms	year built	price
1	1000	1	2001	169900
2	1664	2	2010	368500
3	1268	3	2008	259900
4	3031	4	2003	599000
5	4478	5	2000	699900

TABLE II
RELATIVE GROWTH

$LearningRate$	$Iteration$	$Regularization$		
		0	1	5
0.1	1000	7.03%	8.51%	15.83%
0.1	10000	6.42%	8.47%	15.83%
0.1	100000	6.01%	8.47%	15.83%
0.01	1000	8.18%	9.57%	16.04%
0.01	10000	7.03%	8.51%	15.83%
0.01	100000	6.42%	8.47%	15.83%
0.001	1000	47.84%	48.38%	50.48%
0.001	10000	8.18%	9.57%	16.04%
0.001	100000	7.03%	8.51%	15.84%

2) Results and Evaluation: I tested 5 samples(Table I) to get the predicted house price, with different configuration of $LearningRate$, $Iteration$ and $Regularization$.

I defined the term $RelativeGrowth$:

$$RelativeGrowth_{i,j,k} = \frac{\sum_{s=1}^{5} EachRelativeGrowth_{i,j,k,s}}{5}$$

and

$$EachRelativeGrowth_{i,j,k,s} = \frac{Growth_{i,j,k,s}}{OriginalPrice}$$

$$Growth_{i,j,k,s} = Abs(PredictPrice_{i,j,k,s} - OriginalPrice)$$

Here, $OriginalPrice$ is the lable data of the samples.

$PredictPrice_{i,j,k,s}$ is the predicted price under $LearningRate\ i$, $Iteration\ j$, $Regularization\ k$ and sample s.

$Growth_{i,j,k,s}$ is the absolute growth of the $PredictedPrice_{i,j,k}$ to the $OriginalPrice$ under $LearningRate\ i$, $Iteration\ j$, $Regularization\ k$ and sample s using function $Abs()$.

Table II and Figure 11 shows all the $RelativeGrowth$ of the samples.

From Table II and Figure 11 we can see:

- With the same $LearningRate$ and $Regularization$, $RelativeGrowth$ decreases when $Iteration$ increase.
- With the same $LearningRate$ and $Iteration$, $RelativeGrowth$ increases when $Regularization$ increase.
- With the same $Regularization$ and $Iteration$, $RelativeGrowth$ increases when $LearningRate$ decrease.
- If $Regularization$ is high, then $RelativeGrowth$ will tend to be the same, which means the model is too general

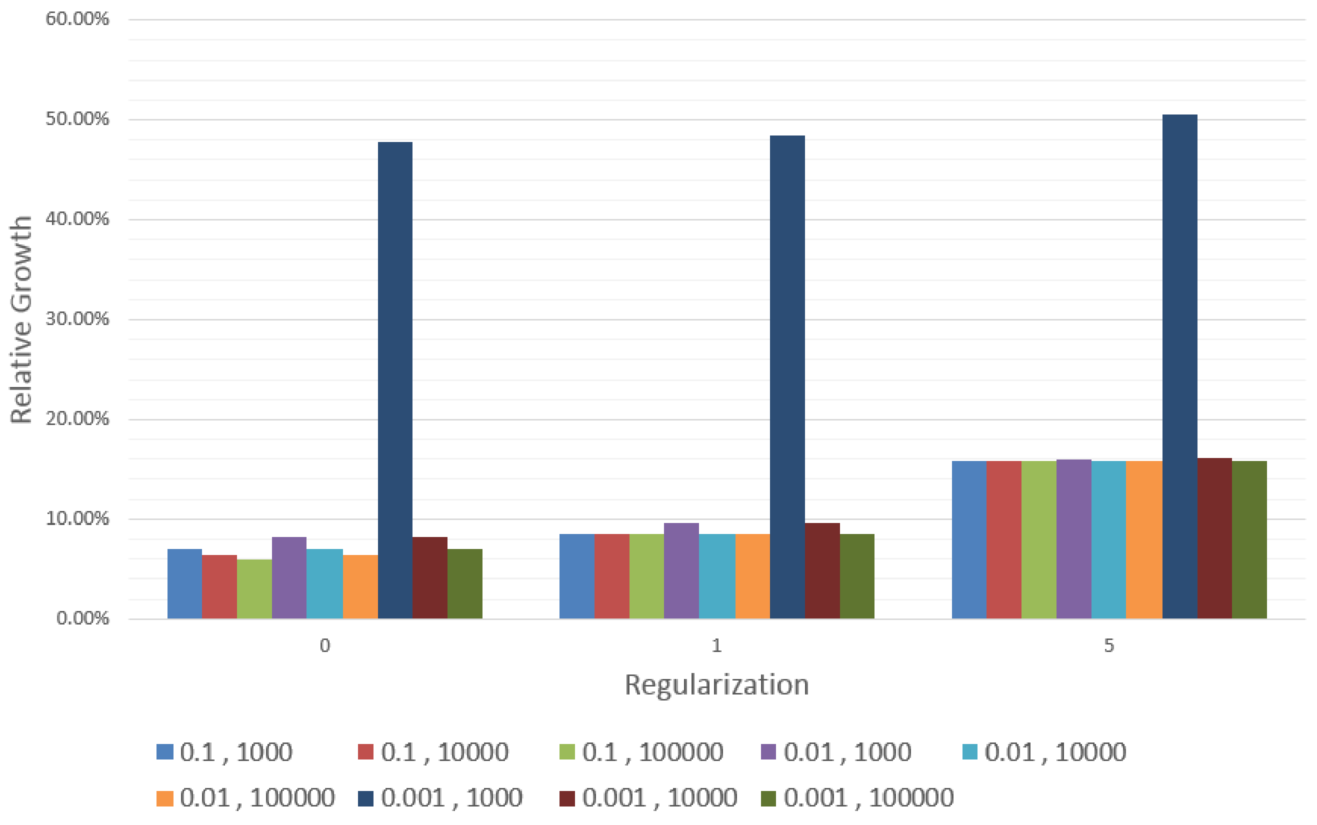


Fig. 10. Relative Growth Graph

and the prediction ends up with under fitting, so that $RelativeGrowth$ will be greater.

- If $Regularization$ is small, for example, 0, although $RelativeGrowth$ is small, but the model may be too specific and the prediction ends up with overfitting.
- When $LearningRate$ is 0.001 and $Iteration$ is 1000, the $RelativeGrowth$ is dramatically large, which is so obvious in Figure 11. However, when $Iteration$ grows, its $RelativeGrowth$ quickly drops.

3) Comment and Assessment: Due to the cockamamie operations of the application's user interface, I just chose 5 typical samples, which can represent most parts of the source data, to get the results. If more samples are added, I believe the result can be more resonable.

The smallest $RelativeGrowth$ is 6.01% here, I think it's a pretty good result, meaning that my model and the configuration of $Regularization$, $Iteration$ and $LearningRate$ are both reasonable and reliable.

However, if I have a better GPU and a faster CPU, I believe I can continue increasing $Iteration$ and decreasing $LearningRate$, so that $RelativeGrowth$ should drop more. This is because larger $Iteration$ and smaller $LearningRate$ will generate a finer model, which will fit the source data better.

In terms of the parallelism performance, this application runs really fast, although I didn't write another program using serialism to compare with the parallelism due to limited time and that I am really unfamiliar with C#.

It's a feature that when $Iteration$ increases, the time needed to complete the prediction will be longer. The reason is that the implementation of the $Iteration$ part of the source program is serial, not parallel.

C. Comparison

As I said before, I didn't write a program running serially. So that I can only make a theoretical comparison between the parallel program and the serial program.

1) Parallelism Statistics & Assumption: The parallel program was divided into two parts, one consisted of normalization, training and prediction, the other was the for loop in $Learn$ method. Most part of the first one was implemented in parallel, however the second part was serial.

Assume now we have got a source data which contains N samples, each of which has 7 features like we've talked before. The test data contains M samples. $Iteration$ is I. And anoter assumption is that the time needed to execute a task by a GPU thread and CPU is the same ,for the convenience of comparison.

Through out the way that a data was trained then prediction was made, we will execute following parallel procedures:

- $MeanFunctor$(8 threads)
- $STDFunctor$(8 threads)

- $FeatureNormalizationgFunctor$($8N$ + $8M$ threads, $8N$ in normalization, $8M$ in prediction)
- $TrainFunctor$(I times, each time N threads)
- $TrainFunctor2$($8I$ times, each time N threads)
- $PredictFunctor$(M threads)

2) Approximation: We also assume that each functor above will execute all its threads at a time unit T at a time. And the other time used in serial computation was T_s.

Now we can calculate the total time needed to perform a prediction using GPU:

$$\begin{aligned} T_{GPU} &= T + T + 2T + IT + 8IT + T + T_s \\ &= (5 + 9I)T + T_s \end{aligned}$$

Then we can approximate the time needed to perform a prediction only using CPU:

$$\begin{aligned} T_{CPU} &= 8T + 8T + 8NT + 8MT + INT \\ &\quad +8INT + MT + T_s \\ &= (16 + (8 + 9I)N + 9M)T + T_s \end{aligned}$$

3) Comparison Between T_{GPU} And T_{CPU}: Now let's use the origial source data and test data in this report(using the best performance configuration) to make a comparison between T_{GPU} and T_{CPU}:

so we have N = 47, M = 5, I = 100000.

$$T_{GPU} \approx 9 \times 10^5 T + T_s$$

$$T_{CPU} \approx 4.23 \times 10^7 T + T_s$$

Assume that the T_s can be ignored, we have:

$$SpeedUp = \frac{T_{CPU}}{T_{GPU}} \approx 47$$

Here we can see that with the increasing of N, $SpeedUp$ can be incredibly great, due to the implementation of parallelism.

IV. Conclusion

GPGPU, Machine Learning and Big Data are three rising fields in the IT industry. There is so much more about these fields than what Im providing in this report. As much as I get deeper into these fields I figure out how well they fit together. From this project I've got some basic idea and a perspective how I can use NVIDIA CUDA easily on machine learning problems. As in any other software solution this example is not the only way to do polynomial regression on house price prediction with GPUs. In fact an enhancement would be supporting multiple GPUs and splitting down the data set into more parts.

References

[1] Nvidia, *THRUST QUICK START GUIDE*, DU-06716-001_v6.5, 2014.
[2] John Owens, David Luebke *Intro to Parallel Programming*, Udacity.
[3] Andrew Ng, *Machine Learning*, Stanford University.