BENCHMARK SUITES FOR STRESS TESTING GPU PROGRAMS

DUAL DEGREE PROECT REPORT

GUIDE: Prof. Rupesh Nasre

Pranav Nair
CS12B062

ABSTRACT

Modern heterogeneous systems are evolving toward a future of intriguing computational possibilities. GPU programming is an important aspect when it comes to heterogeneous parallel programming. Whilst GPU programming can be utilized for higher efficiency and performance, there are a number of factors that come into play which tend to affect the performance of parallelized code drastically and even in some cases prove to be worse off in performance in comparison to serial code. We look at some of the key factors which significantly affect the impact of parallelizing programs and compare their performances and execution times over a range of input parameters. In the second part of this project, we also try to observe and analyze how these factors perform in combination with each other, and try to assert a relation as to which of these factors have a bigger impact towards performance of GPU applications.

Languages and Libraries required:

The CUDA Toolkit focuses on applications whose control part is executed as a process on a general purpose computing device, and which uses one or many NVIDIA GPUs as coprocessors for enhancing the performance of *single program, multiple data* (SPMD) parallel jobs. Such jobs are self-contained, in other words they can be executed and completed by a group of GPU threads totally without intervention by the CPU or the host process, and as a result gaining optimizations from the parallel graphics hardware.

The code which we run on the GPU is implemented as a collection of functions called *kernels* in a language that is basically in C++, but with some extra annotations for distinguishing and separating them from the host code, plus annotations for separating the different types of data memory that are existing on the GPU. These functions may have one or more parameters, and are generally called using a syntax that is highly similar to calling regular C functions, but extended for being able to specify the matrix of GPU blocks and threads that must execute the called function.

The programs are executed on the *libra* cluster for various parameters and results are plotted against the time taken for execution. Plotting of results is done in python using *matplotlib* library.

INTRODUCTION

The CUDA programming model is a heterogeneous model where both the GPU and CPU are utilized for program execution. In CUDA, the *host* refers to the CPU, while the *device* refers to the GPU. Each of these has memory of their own. Code which runs on the host can manage memory on both the host and the device, and also launch *kernels* which are the functions executed on GPU i.e. device. These kernels are basically executed by many GPU threads in parallel to each other. Given the heterogeneous nature of the CUDA programming model, a common sequence of operations for a CUDA program is as follows:

- Declare and allocate host and device memory.
- Initialize host data.
- Transfer data from host to device.
- Execute the kernels.
- Transfer the results from device to host.

An example CUDA kernel would look like the following:

```
__global__ void kernel(int n, float p, float *a, float *b)
{
   int  = blockIdx.x*blockDim.x + threadIdx.x;
   if (k < n) y[k] = p*a[k] + b[k];
}
```

Memory is allocated on device using *cudaMalloc* function present in the CUDA runtime API. The host and the device in CUDA have their own unique memory spaces, both of which can be managed from host code. Memory transfer between device and host is facilitated using *cudaMemcpy*, which works similar to the standard C *memcpy* function, except that it consists of a fourth argument which specifies the direction of the copy. *cudaMemcpyHostToDevice* is used to transfer data from host to device and *cudaMemcpyDeviceToHost* to transfer data in the opposite direction.

This project focuses on implementing Benchmark suits for stress-testing different performance aspects of a GPU program. The main criteria under scrutiny are the following:

- Thread Divergence
- Memory coalescing
- Synchronize atomic operations
- Barriers
- CPU-GPU data flow
- Kernel Count
- CPU intensive operations
- Bank conflicts

We devise programs which stress test the GPU for the aforementioned categories and then evaluate the performance of the same by comparing execution times with the parameterised values used in each code.

1. THREAD DIVERGENCE

Threads from a block are bundled into fixed-size *warps* for execution on a CUDA core, and threads within a warp must follow the same execution trajectory. All threads are supposed execute the same instruction at the same point of time. In other words, the threads cannot diverge. The most common code construct that can cause thread divergence is branching for conditionals in an *if-then-else* statement. If some threads in a single warp evaluate to 'true' and others to 'false', then these 'true' and 'false' threads will clearly branch to different instructions. Some threads will want proceed to the 'then' instruction, while others the 'else'. The main reason for this behavior is because threads in a warp follow SIMT execution mechanism.

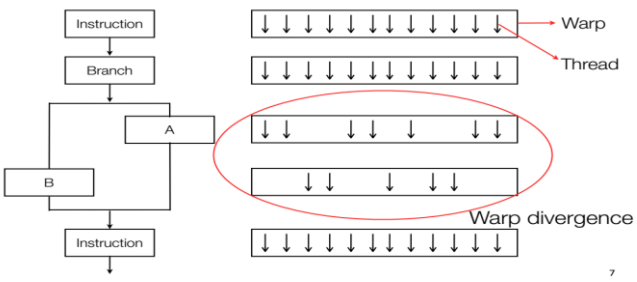


Fig: A basic illustration of thread divergence

As we see from the above figure, when we encounter a branch, some of the threads in a warp have their control flow towards block B and the rest towards block A. But due to the SIMT nature of threads in a warp, the threads executing block B need to wait for the threads executing block A to finish execution before proceeding further hence causing partial serialization of the program.

Following is a basic structure of code which exhibits thread divergence:

```
global__ void dkernel(unsigned *vector, unsigned vectorsize){
  unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
  switch(id){
    case 0: vector[id] = 0; break;
    case 1: vector[id] = vector[id]; break;
    case 2: vector[id] = vector[id - 2]; break;
    case 3: vector[id] = vector[id + 3]; break;
    case 4: vector[id] = 4 + vector[id]; break;
    case 5: vector[id] = 5 - vector[id]; break;
    case 6: vector[id] = vector[6]; break;
    case 7: vector[id] = 7; break;
    case 8: vector[id] = vector[id] + 8; break;
    case 9: vector[id] = vector[id] * 9; break;
  }
}
```

Here we see that the conditional used is a switch statement, which consists of 9 cases. As a result the threads in the warp can execute any one of these 9 cases, and hence there will be a waiting time associated with each group of threads executing one condition and the execution happens sequentially.

A warp contains 32 threads, as a result stress testing would mean using conditionals which diverge as much as possible, worst case in our case being 32 times. A switch statement with 32 cases would ideally satisfy our condition. We check the thread divergence performance with varying number of blocks and also by varying the switch condition and plotting the results.

To assess performance measures for branch divergence, we use a series of methods. First method involved is to parameterize the number of blocks used and the number of threads associated per block. We also parameterize the condition of the switch statement, and compare performance measures. One simple example to parameterize the condition is to use the modulus operator, such as *switch(threadIdx.x%n)* and then vary n over a range of values to compare performance results. We can then plot these versus time taken for execution. Another method which can be used is to parameterize the number of switch statements itself, and this will involve code generation during runtime such that the number of switch statements can be generated as desired by the user.

2. MEMORY COALESCING

Coalesced memory access refers to combining multiple memory accesses into a single transaction. In most modern day GPUs, every successive 32 single precision words (128 bytes) can be accessed by a warp i.e. 32 consecutive threads in a single transaction. However, the following conditions may result in a non-coalesced load, i.e., memory access becomes serialized:

- Memory is not sequential
- Memory access is sparse.
- Misaligned memory access.

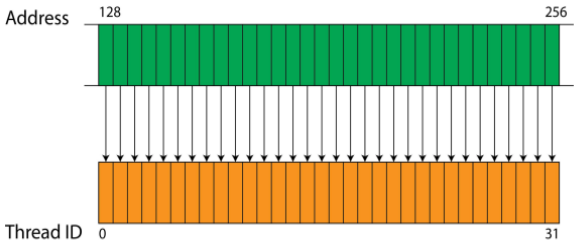


Fig 2.1: Aligned and consecutive memory access

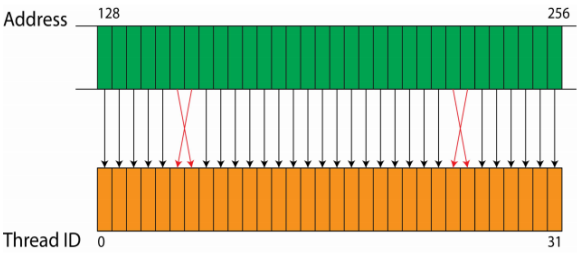


Fig 2.2: Non-consecutive memory access

In fig 2.1, we observe that 32 consecutive threads access 32 consecutive words. The memory access is sequential and aligned, and is therefore coalesced. Whereas in fig 2.2, we observe that memory access is not consecutive, as a result it is uncoalesced.

```
__global__ void offset(T* a, int s)
{
   int i = blockDim.x * blockIdx.x +     threadIdx.x + s;
   a[i] = a[i] + 1;
}

__global__ void stride(T* a, int s)
{
   int i = (blockDim.x * blockIdx.x + threadIdx.x) * s;

   a[i] = a[i] + 1;
}
```

We implement kernels (as shown above) wherein we access memory from a matrix with preloaded values in row major manner and also in a randomized (*strided*) manner and compare the performance values of both these approaches.

The aforementioned kernels are used for *sequential* access and *strided* access respectively. On running these continuously or a million times in a loop we get the execution time for offset kernel as 40 milliseconds and the stride kernel as 50 milliseconds. We clearly observe the improved performance of sequential memory access over the non-sequential approach.

3. SYNC-ATOMIC OPERATIONS

Atomic operations are operations which are performed without interference from any other threads. Atomic operations are often used to prevent race conditions which is one of the common problems prevalent in multithreaded applications. We implement a program where multiple threads try to change the value present at the same memory location. To illustrate the issue at hand, consider two threads A and B. Suppose each thread wants to increase the value of memory location 0x1234 by one. Let us assume that the value at memory location 0x1234 is 5. If we consider that A and B both want to increase the value at location 0x1234 at the same point of time, each of these threads will first have to read the value. Depending on when these reads occur, it is entirely possible that both A and B will read a value of 5 and as a result it gets incremented only once.

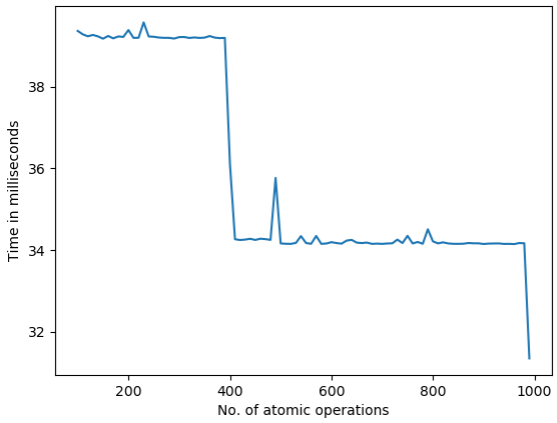
To assess the performance of sync atomic operations, we parameterize the code such that the number of such operations are varied based on the parameter and we evaluate the time taken to execute the same in each case. We plot a time versus number of operations graph to analyze the performance.

Following is a basic atomic operation code:

```
__global__ void dkernel( int * d_in){
   int totalSum;
   if (threadIdx.x == 0) totalSum = 0;
   __syncthreads();

   int localVal = d_in[threadIdx.x];
   atomicAdd(&totalSum, 1);
   __syncthreads();
}
```

We run the code with atomic operations ranging from a 100 to 1000 operations with 10 step increments and plot the same versus time in milliseconds.



The results obtained are quite interesting and unexpected, and it is unclear why we obtain the graph as observed. Further work is required to fully understand this behavior.

4. BARRIERS

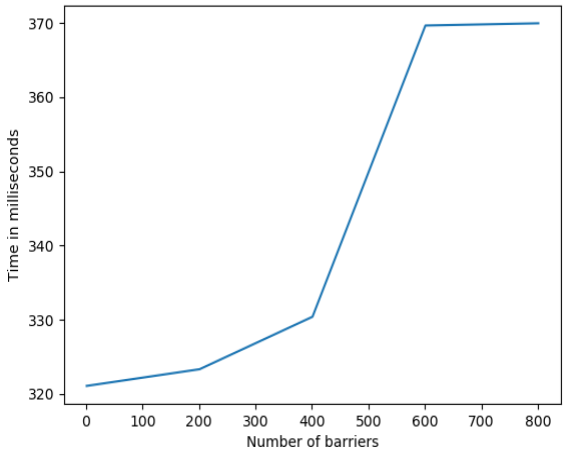
A *barrier* is a thread synchronization construct which when used at a point in the code, is called by each and every thread reaching that point, such that no thread makes progress beyond this construct until all threads (of the team) have arrived at this synchronization point. Once all threads have reached the barrier, they can proceed to execute next sequence of instructions. A simple barrier implementation is as follows:

```
__global__ void dkernel(unsigned *vector, unsigned vectorsize) {
  unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
  vector[id] = id;
  syncthreads();//barrier here

  if(id< vectorsize-1 && vector[id+1]!=id+1)
    printf("Incorrect\n");
}
```

All the threads under execution wait for each other when they encounter the *syncthreads()* statement, and proceed once all the threads have reached this point.

Performance evaluation in the case of barriers has been done by parameterizing the number of barriers used versus execution time. Also to keep the number of threads running higher than the number of barriers in use, the number of threads and blocks spawned is treated as a function of the parameter given. We can predict the increasing nature of the graph as is evident from the following observations.



We observe that for higher values, execution time tends to become stable. This may be due to redundancy in the (high) number of barriers as a result of which waiting time does not increase since the threads are already in sync.

5. CPU-GPU DATA FLOW

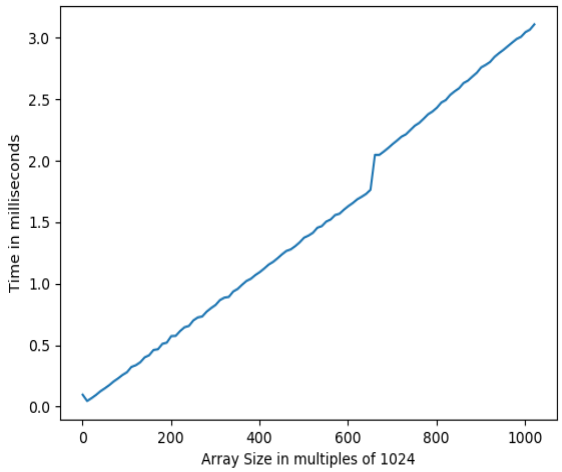
Before executing a kernel on the GPU, all the data required by the kernel needs to be transferred from the host memory to the device memory. After execution, the data produced by the kernel, which contains the results, needs to be transferred back to the CPU memory. The function typically used to accomplish this goal of memory transferring is *cudaMemcpy*.

Increase in data flow from the host to device has a significant impact on the performance of the program. One should always consider the overheads involved in memory transfer before considering the optimizations possible by parallelizing the same using GPU.

To assess the performance of data transfer between the host and device, we implement a program where we using *cudaMemcpy* to transfer a large array of integer values from host to device and vice versa and compare the execution times while varying the amount of data transferred.

We vary the array size in multiples of 1024. The time in milliseconds taken to complete the execution of the program is mostly linear in comparison to the array size. For very large ranges of data (from 0 MB to 2 GB) it is observed that the execution times follow a logarithmic pattern.

Following is the plot obtained for data ranging till approximately 4 Megabytes. As expected we observe a linear plot.

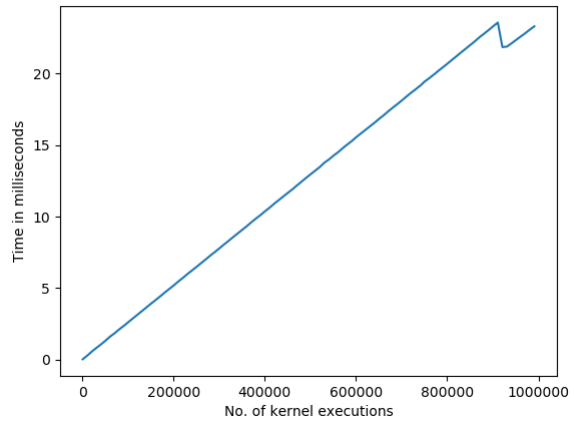


6. KERNEL COUNT

When a GPU program consists of multiple kernels, obviously it would take a longer time to finish execution. In this category we aim to stress test using multiple kernels and compare the performances using this as a parameter.

```
//we run  the kernel in a loop which runs i times.
  for(int i=0;i<N;i++)
    dkernel<<<nblocks, BLOCKSIZE>>>(vector, N,i);
```

The above code fragment is just a depiction of a kernel with *nblocks* number of blocks and *BLOCKSIZE* number of threads. It runs inside a loop which is parameterized to assess performance.

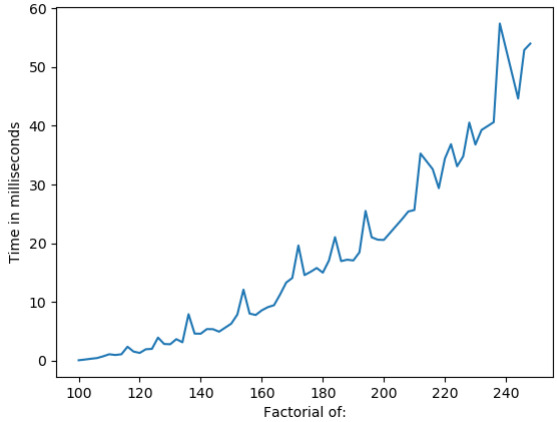


The number of kernel executions is varied from 1024 up to a million in increments of 10000. The time is measured in milliseconds. The graph obtained is linear and it is intuitively clear why this is the case.

7. CPU INTENSIVE OPERATIONS

In this part focus is on CPU intensive kernel codes for stress testing. We look for computationally intensive procedures for performance analysis. The operation chosen here is to calculate factorial of large numbers which includes calculating, multiplying and storing a large amount of digits which cannot normally be stored in the primitive data types such as int or double. It is quite CPU intensive and involves a lot of arithmetic operations which is the main reason for this choice.

For our analysis, we consider varying the number given as input whose factorial is to be found and found the time taken to compute factorial for the same. These values were plotted.



We observe a steady increase in the execution time with respect to the input given. The non-linearity can be attributed to the fact that as the number increases, the data required to store its factorial also increases hence more data transfer from host to device and vice versa. A number of computational factors may also be cited as the reason for such behavior but they are generally trivial in comparison to the memory overhead.

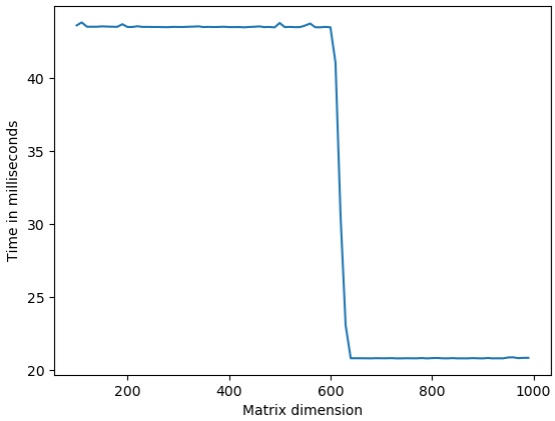
8. BANK CONFLICTS

Bank conflict is a case when any memory access pattern fails to distribute IO across banks available in the memory system. Consider a two dimensional 32x32 array of integers and a DRAM or memory system with 32 banks. By default the array data will be layout in a way that arr[0][0] goes to bank 0, arr[0][1] goes to bank 1, arr[0][2] to bank 2, ...arr[0][31] goes to bank 32. To generalize arr[x][y] occupies bank number y. We consider code that starts accessing data in column major fashion i.e. changing x

while keeping y constant, then the end result will be that all consecutive memory access will hit the same bank, hence causing bank conflict. The following kernel exhibits bank conflict:

```
__global__ void kernel(int* d_A, int pitch)
{
   for (int c = 0; c < height; ++c) {
      for (int r = 0; r < width; ++r) {
         int* row = (int*)((char*)d_A + r * pitch);
         row[c] = row[c]*row[c];
      }
   }
}
```

To assess performance of this kernel, we parameterize the size of the matrix and obtain performance results for matrix dimensions ranging from 100 to 1000. Note that the matrices under consideration are square matrices.



Again we observe very interesting results as observed in the plot. One can speculate that this behaviour could be due to caching effects such as false sharing and so on. Further research is needed on the same to analyse and identify the exact reason for this behaviour.

PART II: JAN-MAY 2017 SEMESTER

The second part of this project involves trying out combinations of the above categories and understand their behavior when they are present simultaneously in a program. There are a total of 28 possible combinations that can be achieved, but one of the key points to note here is that not all of these are feasible and some of them are quite redundant. For example, memory coalescing and bank conflicts are totally different memory related operations. One works on shared memory and the other on global memory (both on the GPU) and as a result we cannot analyze their simultaneous behavior. Barriers are synchronize atomic operations are more or less similar owing to the fact that a sync atomic operation is a barrier of some sort, hence analyzing just one of the two in combination with the rest would be sufficient. Some cases are meaningless to be combined and analyzed as they are independent of each other. For example kernel count versus bank conflicts or any memory operation. The only way to combine these two is to just run the kernel related to the data operations the required number of times, which feels redundant and trivial. Such conflicts and similarities bring down the total number of cases to 16. Analysis is done on these 16 cases separately. 3 dimensional plots are constructed to understand their behavior versus time. X and Y coordinates consist of 2 of the cases under consideration, and these are varied and then the execution time is plotted. Following cases are analyzed and their behavior is plotted versus time:

- Thread divergence vs barriers
- Thread divergence vs kernel count
- Thread divergence vs CPU intensive operations
- Thread divergence vs memory coalescing
- Thread divergence vs CPU-GPU data flow
- Thread divergence vs bank conflicts
- Barriers vs CPU intensive operations
- Barriers vs CPU-GPU data flow
- Sync-atomic vs memory coalescing
- Sync atomic vs bank conflicts
- Kernel count vs barriers
- CPU intensive vs CPU-GPU data flow
- CPU intensive vs bank conflicts
- CPU intensive vs memory coalescing
- Memory coalescing vs CPU-GPU data flow
- CPU-GPU data flow vs bank conflicts

For each of these pairs under consideration, both the categories are varied in their range. For example, when we consider Thread divergence or memory coalescing, we can only have 32 way divergence or 32 way coalescing, since a warp consists of 32 threads which execute in SIMD fashion. For other criteria like kernel count, data flow etc, we consider an appropriate upper limit, say 1 million kernel executions and so on.

THREAD DIVERGENCE VS BARRIERS

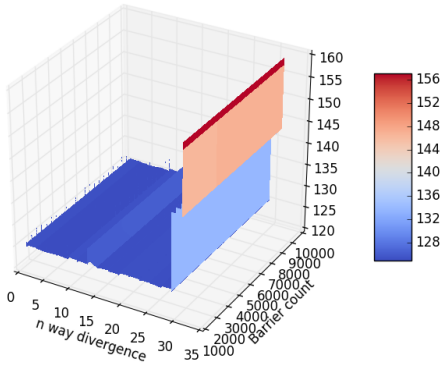
Here we try to combine Thread divergence with barriers and try to analyze the behavior of the program under varying degrees of divergence and barriers simultaneously. The basic idea behind divergence versus the rest of the criteria is relatively straightforward. We use a similar approach as compared to the earlier work, wherein a switch case with 32 cases is used. Then under each of these cases we insert code with barriers, and this can be varied according to how much is required.

Following is a code snippet which highlights the above points:

```
__device__ void barrier(float *vector,int i) {
unsigned id = blockIdx.x * blockDim.x + threadIdx.x;
  vector[id] = id;
  int flag=0;
  for(int g=1;g<=i;g++){
          if(g%2==0)
                  flag++;
          else
                  flag--;
          __syncthreads();//barrier here
  }
}
```

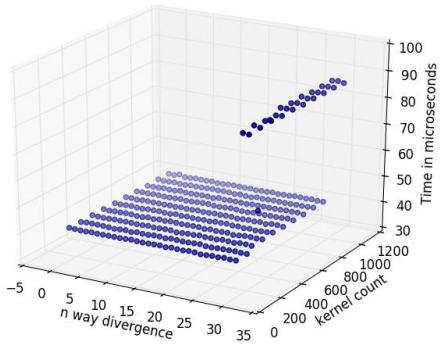
This is the barrier code which is used in the switch statement. As we can see the variable i controls the number of barriers that are required. The thread divergence is controlled using another variable which is used to adjust the number of accessible switch cases accordingly.

The plot shown below is obtained when we vary the parameters for divergence and number of barriers from 1 to 32 way divergence and upto 10000 barriers respectively. It is an interesting behavior, and we observe that beyond 25 way divergence the execution time rises sharply, irrespective of the barrier count.



THREAD DIVERGENCE VERSUS KERNEL COUNT

In this case we vary thread divergence and kernel count and observe the behavior of the same. We use the same approach as above, and instead of barriers we utilize the code for kernel count from earlier work.

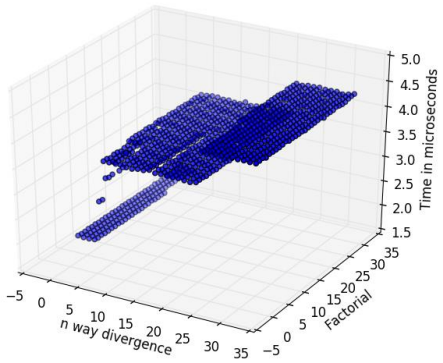


THREAD DIVERGENCE VERSUS CPU INTENSIVE OPERATIONS

Again we use the same proposed method for thread divergence scenario, and for cpu intensive operations, we use code for calculating factorial for large numbers. So the switch cases essentially contain the factorial code, and in this case we vary the factorial required which is depicted on the Y-axis. We use the following _device_ method for CPU intensive code, which is called in the switch statement.

```
__device__ void dkernel(int *res,int n,int *res_size) {
//   Initialize res
   res[0] = 1;
//   int res_size = 1;

   for (int x=2; x<=n; x++){
      int carry = 0;  // Initialize carry
      for (int i=0; i<*res_size; i++)
      {
         int prod = res[i] * x + carry;
         res[i] = prod % 10;
         carry  = prod/10;
      }
      while (carry)
      {
         res[*res_size] = carry%10;
         carry = carry/10;
         (*res_size)++;
      }
   }
}
```



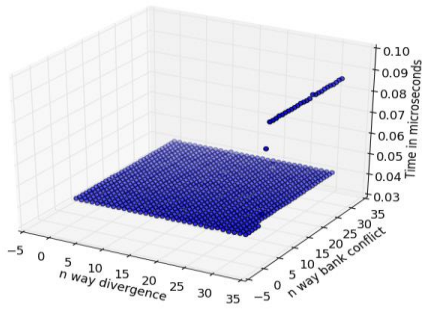
In the plot we observe that as the divergence increases the time also increases but the increase is not steady, rather in steps. This is an unusual behavior and needs to be further investigated.

THREAD DIVERGENCE VS COALESCING

For the case of memory coalescing, we try to vary the level of coalescing for different degrees of divergence. This is achieved by setting the appropriate stride required. For instance, setting the stride to 1 results in a perfectly coalesced memory access, and setting the stride to 32 results in the most uncoalesced memory access. In the

plot, the stride is depicted on the Y axis. The following function is utilized for coalescing operation, and it is clear that the parameter val2 is used as the stride which controls the degree of coalescing in this _device_ function.

```
__device__ void coal(float * d_out, float * d_in,int val2){
        float f;
        for(int i=0;i<1024;i+=val2)
        {
                f= d_in[i];
                d_out[i]=f*f;
        }
}
```

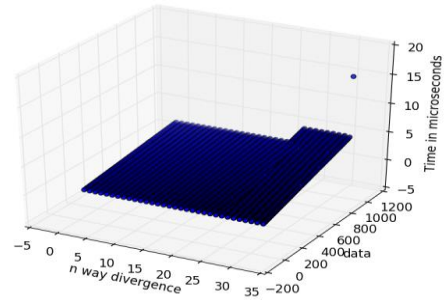


DIVERGENCE VERSUS DATA FLOW

Varying data flow is quite straightforward as we just need to transfer more data from CPU to GPU and vice versa. Again this transfer is done in varying proportions with varying degrees of divergence. The following code snippet shows how data flow is varied in the program.

```
unsigned vec[ARRAY_SIZE*j];
for (int i = 0; i < 1024*j; i++)
  vec[i] = i;
cudaMalloc(&vector, ARRAY_SIZE*j * sizeof(unsigned));
cudaMemcpy(vector,vec,ARRAY_SIZE*j*sizeof(unsigned),cudaMemcpyHostToDevice);
```

It is evident from the above snippet that j is a parameter that is used to assign the required size to the array. Using this parameter and the parameter for divergence we generate the plot.



It is observed that there is a steady increase in both directions, for higher data flow as well as higher divergence the execution time increases. But it can also be seen that the rate of change of time with data flow overshadows the change in time with respect to change in divergence. Hence it can be concluded that data transfer has a larger impact in a way on programs compared to thread divergence.

DIVERGENCE VERSUS BANK CONFLICTS

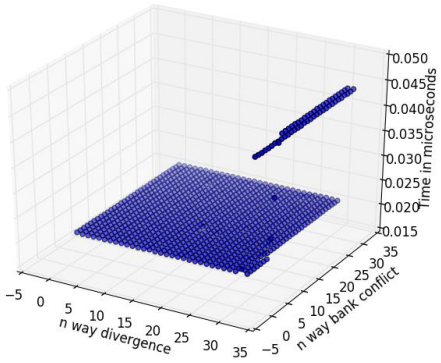
GPUs have a special fast memory for sharing the current working set among threads executed at the same time. This fast memory is called shared memory. Accesses to shared memory are processed differently. For example, they are not coalesced into larger blocks. Instead, the stored data is sliced into a number of banks, and each bank serves accesses independently, one access at a time. Ideally, different threads access different data in different banks or the same data in the same bank; in the latter case, the data is accessed once and then broadcast. Otherwise, some banks perform several data accesses, which takes an additional time; it is said, then, that the respective warp access causes bank conflicts.

Hence the objective here should be to restrict the number of banks and vary this with respect to divergence and observe the behavior of the program. As seen in earlier work, a 2 dimensional matrix is used for bank conflicts. Restricting the number of banks involves restricting the accessible width in the matrix. Again this is done in the presence of varying degrees of divergence and the results are plotted.

```
__device__ void kernel(int* d_A, int pitch,int height,int width)
{
  for (int c = 0; c < height; ++c) {
    for (int r = 0; r < width; ++r) {
      int* row = (int*)((char*)d_A + r * pitch);
      row[c] = row[c]*row[c];
    }
  }
}
```

Here width is the parameter which is varied as necessary.

On running the code as explained above, from 1 way to 32 way divergence and bank conflict, we obtain the plot as shown below. Further analysis is needed to understand the behavior of this plot as to why the execution time remained constant for the given program irrespective of the parameter changes over almost the entire range of values.



BARRIERS VS CPU INTENSIVE OPERATIONS

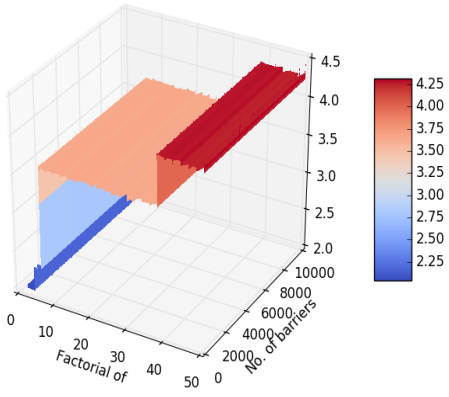
The same code is used for CPU intensive operations as earlier, except in this case we add barriers to this code and this barriers can be varied to the required degree using a parameter. Following is the code snippet of the kernel being used:

```
__global__ void dkernel(int *res,int n,int *res_size,int v2) {
  // Initialize result
   res[0] = 1;
  // int res_size = 1;

  for (int x=2; x<=n; x++){
     int carry = 0;  // Initialize carry
     for (int i=0; i<*res_size; i++)
     {
        int prod = res[i] * x + carry;
        res[i] = prod % 10;
        carry  = prod/10;
     }
     while (carry)
     {
        res[*res_size] = carry%10;
        carry = carry/10;
        (*res_size)++;
     }
  }

   int flag=0;
   for(int g=1;g<=v2;g++){
            if(g%2==0)
                     flag++;
            else
                     flag--;
            __syncthreads();//barrier here
   }
}
```

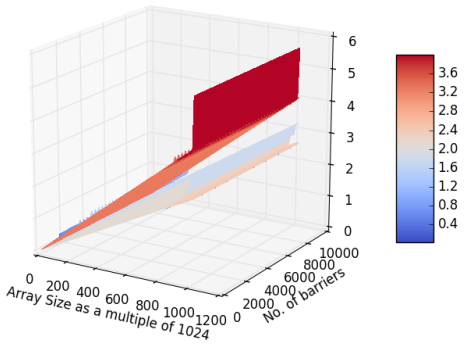
The variable v2 which is a parameter is used to vary the percentage of barriers required. The other parameter n is the number whose factorial is to be calculated.



The plot is not a scatter plot unlike earlier examples but rather a surface plot. From the plot it is evident that the execution time is largely impacted by the factorial change and the impact due to number of barriers is insignificant in comparison. This is understandable intuitively, since factorial has a large number of computations and also barriers tend to get redundant after all threads have already been synced in the time the factorial is being computed.

BARRIERS VERSUS DATA FLOW

This is quite a straightforward procedure, we use the data flow kernel from earlier and add barriers to it, both are parameterized and results are plotted versus execution time. The same loop statement is used for barrier as in earlier examples. The following plot is obtained.



This is also a surface plot, and quite an unusual result. We observe that the barrier count is insignificant in comparison to data flow, in terms of impacting the execution time. This behavior is similar to the previous example, and probably a similar explanation exists for the same. We also notice a sharp increase at very high data flow values, this may be because of hardware constraints such as exceeding the global memory in GPU.

SYNC-ATOMIC VERSUS MEMORY COALESCING

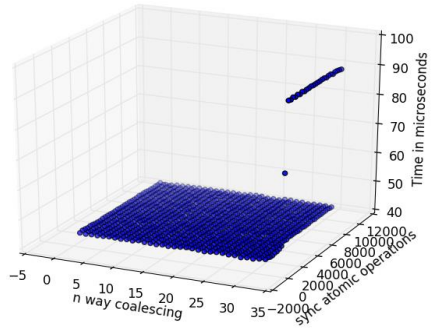
Here we use the code for coalesced memory access, with the difference being that we add sync atomic operations to analyze the behavior when these 2 categories interact. To achieve this, atomic operations are used on the data which is being accessed in the memory coalescing kernel. This way we combine both categories and also use parameters to vary the two as needed and observe the results.

```
__global__ void coal(int * d_out, int * d_in,int val1,int val2){
        int f;
        int totalSum=1;
        for(int i=0;i<1024;i+=val2)
        {
                f= d_in[i];
                d_out[i]=f*f;

        if (threadIdx.x == 0) totalSum = 0;
         __syncthreads();

         for(int i=0;i<val1;i++)
            atomicAdd(&totalSum, d_out[i]);
        __syncthreads();
        }
}
```

The code snippet above shows the function which exhibits coalescing and has an atomicAdd() operation. Both of these are parameterized and varied and the results are obtained.

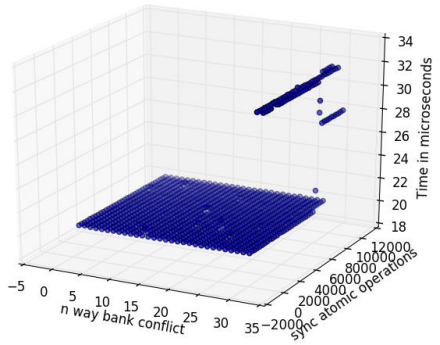


SYNC ATOMIC VERSUS BANK CONFLICTS

This is similar to the case above, except in this case we are considering bank conflicts instead. We use atomic operations on the 2 dimensional matrix in the bank conflict kernel code and observe the results as the parameters are varied.

```
__global__ void kernel(int* d_A, int pitch,int height,int width,int
val1)
{
   int totalSum=1;
   for (int c = 0; c < height; ++c) {
      for (int r = 0; r < width; ++r) {
         int* row = (int*)((char*)d_A + r * pitch);
         row[c] = row[c]*row[c];
         for(int i=0;i<val1;i++)
             atomicAdd(&totalSum, row[c]);
          __syncthreads();
      }
   }
}
```

From the above code we observe that atomicAdd() operation is used on the matrix which is used in the bank conflict code. Both are parameterized as discussed earlier and the observations are plotted.



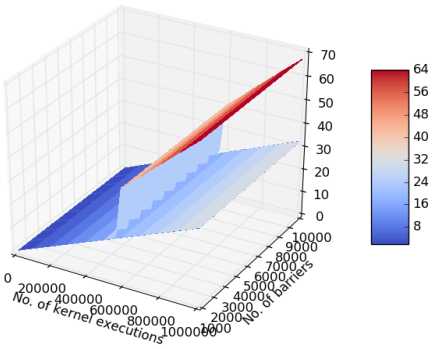
KERNEL COUNT VS BARRIERS

The following code snippet highlights how kernel count is parameterized.

```
for(int v1=10000;v1>=1;v1-=1000){
for(int j=1000000;j>=1;j-=10000) {
   struct timeval  tv1, tv2;
   gettimeofday(&tv1, NULL);
for(int i=0;i<j;i++){
   dkernel<<<nblocks, BLOCKSIZE>>>(vector, N,i,v1);
```

```
  cudaMemcpy(hvector,vector,N*sizeof(unsigned),
cudaMemcpyDeviceToHost);}
  gettimeofday(&tv2, NULL);
  printf ("%d %d %f\n",j,v1,(double) (tv2.tv_usec - tv1.tv_usec) /
1000000 + (double) (tv2.tv_sec - tv1.tv_sec));
  cudaDeviceSynchronize();
 }
}
```

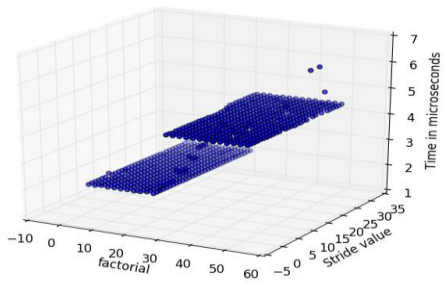
As is evident from the code, the loop which encloses the kernel call has an exit condition which is parameterized. The kernel itself just consists of a simple barrier code which is parameterized using the variable v1. The following plot is obtained.



We see a similar trend yet again, the change in time with respect to barrier count is insignificant compared to the rate of change of time with respect to the kernel count.

CPU INTENSIVE OPERATIONS VS MEMORY COALESCING

For CPU intensive operations we use the factorial code from earlier, but we need to add data operations to implement parameterized memory coalescing into it. This is achieved by using the intermediate result array which is supposed to contain the factorial and we perform operations on this array of values based on the required stride which is the parameter. The plot obtained is shown below.

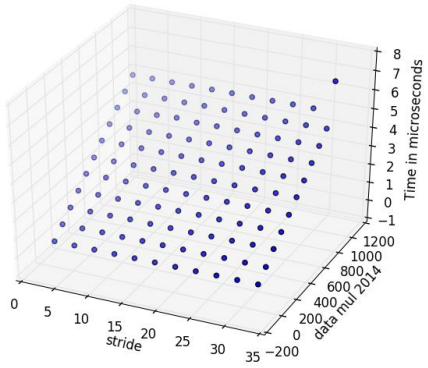


The result obtained is quite unusual and follows a similar pattern to other results involving memory coalescing. We get almost constant execution time values irrespective of the stride but this time there are 2 planes obtained.

MEMORY COALESCING VERSUS DATA FLOW

This case is quite straightforward as we just need to apply the memory coalescing procedure discussed earlier to varying levels of data flow. Hence the only difference is the amount of data that the memory coalescing kernel works on which is varied as shown in the plot and the behavior is analyzed.

Data flow is varied in the main method as we had discussed earlier, and then the same procedure for memory coalescing is used as in the previous cases, except that the data used is the one currently being transferred. These are now varied using parameters and the plot is obtained versus time.



The scatter plot is obtained as shown above. We observe that data variation plays a significant role towards execution time compared to stride change.

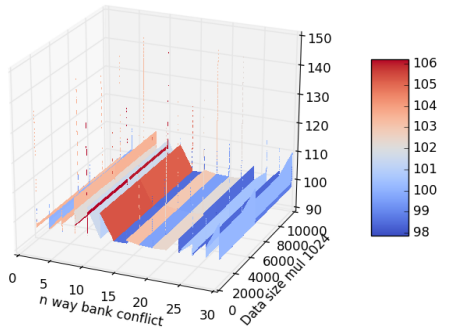
DATA FLOW VERSUS BANK CONFLICTS

As done in the earlier illustrations, the operations done with coalescing can also be done with bank conflicts. So a similar procedure is adopted as above, except in this case it is the matrix dimensions we vary to achieve varying degrees of data transfer from CPU to GPU and vice versa.

```
__global__ void kernel(int* d_A, int pitch,int height,int width)
{
  int totalSum=1;
  for (int c = 0; c < height; ++c) {
    for (int r = 0; r < width; ++r) {
```

```
        int* row = (int*)((char*)d_A + r * pitch);
        row[c] = row[c]*row[c];
    }
  }
}
```

In the above kernel code, width is the parameter which controls bank conflict, and height is the parameter which controls the data flow from CPU to GPU.



We obtain the following plot as shown above, which is a rather unusual and unexpected behavior. Further analysis is needed to understand this behavior. This also conforms with the unexpected behavior observed with most memory related operations i.e. memory coalescing and bank conflicts in particular.

FUTURE WORK

PART I

The results obtained for the synchronize-atomic operations and bank conflicts is not intuitively clear and is only speculative at the moment. Further analysis is needed in this area to assess the performance result for these aspects. Also using code generation is an important criteria to assess performance in the case of thread divergence, which will be an area of focus in the future work.

Also future work consists of designing tools for measuring degree of various optimizations in an input program, so that one could find out an exact measure for thread divergence, memory coalescing etc. in his program and have an estimate of how much optimizations can be performed on the same.

PART II

Some of the plots, especially in the cases involving memory coalescing, show rather constant execution times for various degrees of coalescing. This requires further investigation this was not the expected behavior we were looking for and it is unclear at this moment why this is the case. Also future work can involve analysis of behavior of more than 2 categories taken at a time, for example how would a code behave under simultaneous instances of divergence, coalescing and high data flow etc., with all these being programmed as dependent operations.

REFERENCES

[1] P. Bialas and A. Strzelecki: Benchmarking the cost of thread divergence in CUDA.

[2] Vasily Volkov,Understanding Latency Hiding in GPU's.

[3] Naznin Fauzia, Louis-Noel Pouchet and P. Sadayappan: Characterizing and Enhancing Global Memory Data Coalescing on GPUs.

[4] David Kirk and Wen Mei-Hwu: Programming Massively Parallel Processors.

[5] Gert-Jan van den Braak, Bart Mesman, Henk Corporaal: Compile-time GPU Memory Access Optimizations.

[6] Diego Alejandro Rivera-Polanco: Collective Communication and Barrier Synchronization on NVIDIA CUDA GPU.

[7] Mark Harris: High performance computing with CUDA.

[8] Jim Demmel, UC Berkeley: Designing fast linear algebra kernels in the presence of memory hierarchies.

[9] Tianyi David Han Tarek S. Abdelrahman: Reducing Branch Divergence in GPU Programs.

[10] CUDA documentation provided at https://developer.nvidia.com/accelerated-computing-training

[11] The supercomputing blog. http://supercomputingblog.com/cuda-tutorials/