Communication Optimization for Multi GPU Implementation of Smith-Waterman Algorithm

Sampath Kumar
NVSSP
Department of Mathematics and Computer Science
Sri Sathya Sai Institute of Higher Learning
Prasanthi Nilayam, A.P.-515134, India

P. K. Baruah
Department of Mathematics and Computer Science
Sri Sathya Sai Institute of Higher Learning
Prasanthi Nilayam, A.P.-515134, India

ABSTRACT
GPU parallelism for real applications can achieve enormous performance gain. CPU-GPU Communication is one of the major bottlenecks that limit this performance gain. Among several libraries developed so far to optimize this communication, DyManD (Dynamically Managed Data) provides better communication optimization strategies and achieves better performance on a single GPU. Smith-Waterman is a well known algorithm in the field of computational biology for finding functional similarities in a protein database. CUDA implementation of this algorithm speeds up the process of sequence matching in the protein database. When input databases are large, multi-GPU implementation gives better performance than single GPU implementation. Since this algorithm acts upon large databases, there is need for optimizing CPU-GPU communication. DyManD implementation provides efficient data management and communication optimization only for single GPU. For providing communication optimization on multiple GPUs, an approach of combining DyManD with a multi-threaded framework called GPUWorker was proposed. Our contribution in this work is to propose an optimized CUDA implementation of this algorithm on multiple GPUs i.e., GPUWorker-DyManD which reduces the communication overhead between CPU and multiple GPUs. This implementation combines DyManD functionality with GPUWorker for optimizing communication. The performance gain obtained for the GPUWorker-DyManD implementation of this algorithm over default multi-GPU implementation is 3.5x.

Keywords
DyManD, GPUWorker, Data Mangement, Communication Optimization.

1. INTRODUCTION
GPUs are capable of accelerating real time applications and can achieve performance gain of hundreds of GFLOPS. Real applications, rewritten to take advantage of GPUs, regularly achieve speedups between 4 and 100x [1]. But CPU-GPU communication is one of the bottlenecks that limit the performance gain that can be achieved by these applications. As a part of communication, manually managing data transfers between CPU and GPU is complex and error-prone. And cyclic communication patterns between CPU and GPU increases the execution latency and reduces the parallelism. So when applications are implemented with efficient data management and optimized communication, the complexities involved in CPU-GPU communication can be eliminated.

Applications like Smith-Waterman require large databases for its execution and hence there is need for optimizing communication for such applications.

1.1 Data Management and Communication Optimization
CPU and GPU have separate memories in a typical CPU-GPU memory architecture. When a program executes on both the host and device, data needs to be transferred between CPU and GPU. Each processing unit efficiently access only its memory and for accessing data-structures outside their memory, the data must be explicitly copied between the divided CPU and GPU memories. This process of copying data between these memories for correct execution is called Communication Management [2]. But, manually managing this communication is laborious and error-prone. The complexities involved with manual transfer of correct data between host and device are pointer aliasing, subversive typecasting, handling complex structures of variable sized arrays and jagged arrays, handling global pointers. The programmer must manage buffers and manipulate pointers which are well-known sources of bugs.

Unfortunately, not all communication management techniques are efficient; because typical GPU implementation consists of cyclic communication patterns. Cyclic communication pattern involves copying data from CPU to GPU memory, launching a GPU kernel, and copying back results from GPU to CPU memory. Cyclic communication patterns prevent the systems from efficiently parallelizing complex programs that launch many GPU kernels and are orders of magnitude slower than acyclic patterns [3]. Transforming cyclic communication patterns to acyclic patterns is called Optimizing Communication. Copying data to the GPU in the pre-header, spawning many GPU functions, and copying the result back to CPU memory in the loop exit yields an acyclic communication pattern. Incorrect communication optimization causes programs to access stale or inconsistent data.

1.2 Smith-Waterman algorithm
This is a most frequently used algorithm in the field of computational biology for implementing local sequence alignment in protein or nucleotide databases [4]. It identifies similar regions between two protein sequences or nucleotide sequences. This algorithm compares subsequences of all possible lengths and optimizes the similarity measure instead of trying to match the total sequence. This algorithm uses dynamic programming and the alignment takes place in a 2D

matrix where each entry represents the pairing of one character from each sequence. Each entry in the matrix contains two values: a score and a pointer. This algorithm consists of three basic steps:

- **Initialization**: The first row and column of the matrix are initialized with zeroes.
- **Fill**: All cells in the matrix are filled with scores and a pointer. To find the score of a cell the maximum value among three scores: a match score, a vertical gap and a horizontal score can be found. So the scores in all the three cells which include left cell, top cell and the diagonal cell are required to calculate the score in the present cell.
- **Trace-Back**: This step recovers the alignment from the matrix. It starts from the bottom-right corner and follows the pointer until it reaches the top-left corner of the matrix.

CUDA implementation of this algorithm is described as follows. Host memory is allocated for protein database and query arrays. Device memory is allocated for both these arrays and the data is transferred to the device using CUDA functionality. Then CUDA kernel is launched on the device for finding best sequence match for the query sequence from the database. Each thread in the kernel grid computes similarity score for one of the protein sequences in the database. Resulting score array is transferred from the device to host. Best score across all the threads gives the best sequence match for the given query. Some of the variants of this implementation give improved performance by optimizing its performance on single and multiple GPUs.

For larger input database sizes, multi-GPU implementation gives better performance than single GPU implementation. In multi-GPU implementation, input protein database is divided between multiple devices and sequence matching is done on individual portions of the database.

Our goal in this work is to provide an optimized multi-GPU implementation of Smith-Waterman algorithm by optimizing communication between CPU and GPUs. Several communication optimization techniques have been developed so far for providing efficient data management and communication optimization. The most efficient technique for optimizing communication between CPU and GPUs for multi-GPU implementation of Smith-Waterman algorithm is proposed in this work.

In Section 2 the existing techniques that provide communication management and communication optimization will be discussed. In section 3 the design and implementation of our approach for communication management for Smith-Waterman algorithm will be discussed. Section 4 evaluates the performance of our approach with respect to default multi-GPU implementation.

2. RELATED WORK

IE (Inspector - Executor) [5], CGCM (CPU - GPU Communication Management) [2] and DyManD (Dynamically Managed Data) [6] are some of the efficient CPU-GPU communication optimization techniques for GPU applications. Among all these techniques, DyManD implementation overcomes the limitations of other techniques and hence is more preferable for communication optimization. The implementation details, advantages and disadvantages of each of these techniques will be discussed here.

IE performs dynamic management of data but does not provide communication optimization. In IE, a compiler creates an inspector for every parallelized region. The inspector loads the data needed by a parallelized region and transfers the data to the appropriate memory space. Parallel functions use this data for execution. IE is best suited for distributed memory clusters and not for GPUs because the communication is cyclic for each GPU function. When IE is applied to GPUs yields a whole program slowdown compared to sequential execution due to this cyclic communication [2].

2.1 CGCM

CGCM is the first fully automatic system for managing and optimizing CPU-GPU communication. Semi-automatic communication techniques that are proposed earlier to CGCM have limited applicability and they lack optimized communication [2]. This technique avoids the limitations of IE with the help of a run-time support library and an optimizing compiler to automatically manage data and to optimize CPU-GPU communication, respectively.

The run-time library is used to determine correctly and efficiently which bytes to transfer from host to device. For maintaining correctness, the run-time library copies data from CPU to the GPU at allocation unit granularity. An allocation unit is a contiguous region of memory allocated as a single unit. The run-time library maintains a self-balancing binary tree map for storing allocation units. The base and size of each allocation unit are stored in the map. The map is indexed by the base address of each allocation unit.

CGCM compiler pass uses the run-time library for managing data transfer automatically. The compiler uses liveness analysis for determining values that are to be transferred to GPU. For each GPU function, the compiler creates a list of live-in values. A value is live-in if it is passed to the GPU function directly or if it is a global variable used by the GPU. For each live-in pointer to each GPU function, the compiler transfers data to the GPU by inserting run-time library calls. After the GPU function call, the compiler inserts a call for each live-out pointer to transfer data back to the CPU.

CGCM compiler uses three passes to optimize communication. All these compiler passes transform cyclic communication patterns to acyclic patterns. Three compiler passes are map promotion, alloca promotion and glue kernels. For the mapped data in a function or a loop, the functionality of Map promotion is to hoist the run-time library calls out of them if the data is not referenced or modified. Map promotion cannot hoist run-time library calls for local variables. The other two passes helps Map promotion in optimizing communication. Alloca promotion takes care of local variables by pre-allocating them in the stack and thus helps in map promotion. Glue kernels pass identifies small CPU code regions between two GPU functions which prevent map promotion and transform that region into single GPU function. Thus alloca promotion and glue kernels compiler passes improve the applicability of map promotion.

Overall implementation of CGCM is shown in the figure 1[2]. By implementing CGCM, a whole program speedup of 5.36x over the best sequential CPU-only execution can be achieved. Though implementation of CGCM provides improved performance, execution of CGCM has its own limitations [2]. CGCM cannot be used for recursive data structures like trees, linked lists etc. The performance of CGCM is limited by the

static analysis (type inference and alias analysis) performed during the execution.

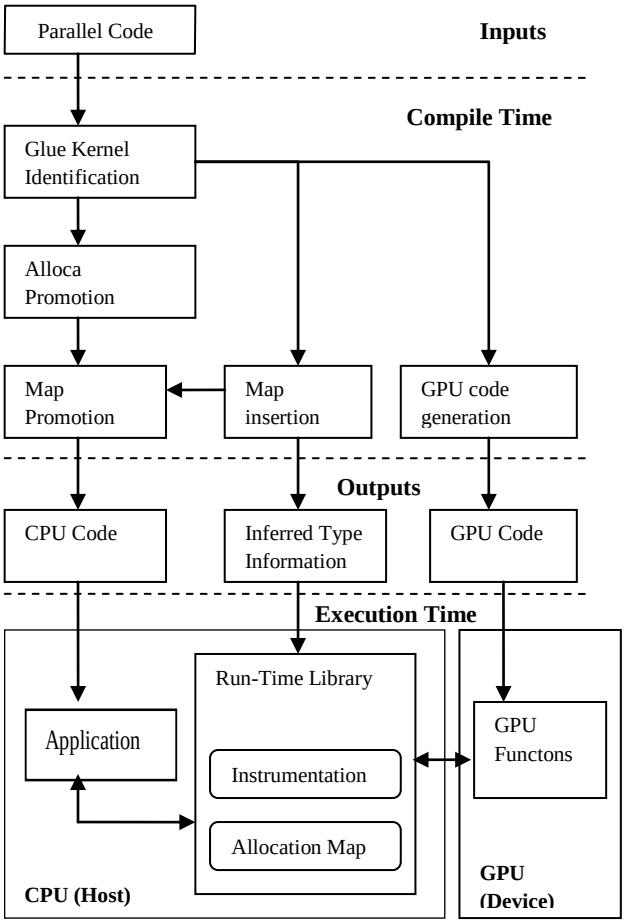

Fig. 1. High Level Diagram of CGCM

2.2 DYMAND

DyManD is another fully automatic system for providing data management and optimizing CPU-GPU communication. It combines dynamic analysis from IE with efficient acyclic communication from CGCM [6]. Thus it addresses the drawbacks of both the methods. DyManD uses dynamic analysis and some heuristics to overcome the limitations caused by static analysis (type inference and alias analysis). For managing transfer of complex and recursive data-structures, DyManD creates the illusion of a shared CPU-GPU memory. DyManD ensures that every allocation unit on the CPU has a corresponding allocation unit on the GPU at the same numerical address. Thus DyManD results in providing direct address translation for allocation units in CPU to equivalent GPU locations and overcomes CGCM's static type inference.

For avoiding static alias analysis DyManD uses page protection system. Using this, DyManD transfers data from GPU to CPU memory only when needed. DyManD removes read and write privileges from the allocation units in CPU memory after copying them to GPU memory. If the CPU accesses the pages later, the program will fault, and DyManD will transfer the affected allocation units back to CPU memory, mark the pages readable and writable, and continue execution. Thus DyManD transfers data between GPU and CPU only when there is need for that data in either host or device and thus avoids cyclic communication to a larger extent.

The DyManD data management and communication optimization system consists of three parts: a memory allocation system, a run-time library, and compiler passes. The memory allocation system transfers data between CPU and GPU at equivalent addresses and thus reduces the burden of translation from the run-time system . The run-time system manages data and optimizes communication at run-time. The compiler inserts calls to the memory allocation system and to the run-time library into the original program, and it generates DyManD compliant assembly code for the GPU.

DyManD memory allocation system allocates two memory blocks, one on host and another on GPU. The two blocks are of same size and same address. Currently, there is no way to allocate memory at fixed GPU addresses. Therefore, it first allocates GPU memory normally and then uses mmap to map a numerically equivalent address in CPU memory. DyManD uses bitmasks to ensure that GPU allocations do not overlap with program's static memory allocations. Static allocations start at low addresses so the allocation system sets a high address bit to avoid overlapping static and dynamic allocations. A bitwise mask operation before each GPU memory access recovers the original GPU pointer.

DyManD's run-time library manages data and optimizes communication. For each allocation unit, at run-time, an ordered map is maintained from the base address to the size and state. The map is used for determining the extent to which a pointer-sized value points to and checking if the range lies within an allocation unit. Allocation unit can be in any of three states: CPU Exclusive (CPUEx), GPU Exclusive (GPUEx) and Shared. All allocation units begin in the CPUEx state. In the CPUEx state, the CPU has exclusive access to the allocation unit. The Shared state signifies that a specific allocation unit and any other allocation units it points to recursively should be copied to the GPU before invoking the next GPU function. Shared allocation units can be accessed on CPU but will become GPUEx on the next GPU function invocation. Accesing any byte in a pro-tected allocation unit triggers an exception. The exception handler copies the allocation unit back to CPU memory and changes the state to Shared. State transitions using DyManD is shown in the figure 2[6].

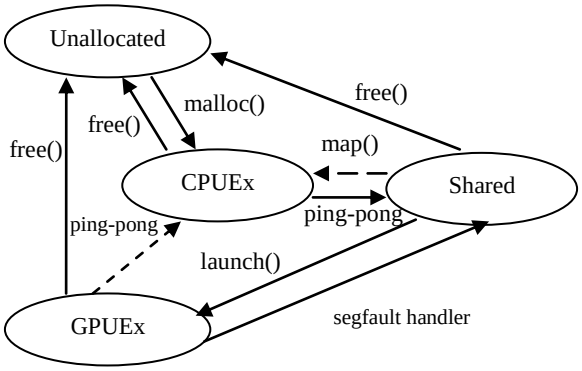

Fig.2. DyManD's state transition diagram

DyManD uses two compiler passes for optimizing communication dynamically: alloca promotion and glue kernels. Both these optimization techniques were same as

defined in CGCM. Thus using all these compiler passes, runtime library and memory allocation system, DyManD effciently manages and optimizes CPU-GPU communication. DyManD outperforms CGCM equipped with production-quality and research grade alias analyses, achieving a whole program geomean speedup of 4.21x over best sequential execution versus geomean speedups of 2.35x and 1.28x, respectively, for CGCM [6].

3. COMMUNICATION OPTIMIZATION ON MULTIPLE GPUs

The main goal of this work is to provide communication optimization framework for multi-GPU implementation of Smith-Waterman. DyManD's implementation provided optimized communication on a single GPU. For providing communication optimization for multiple GPUs, a multi-threaded framework called GPUWorker [7] along with DyManD was used. GPUWorker concurrently launches kernels onto multiple devices. The total overhead of running applications on multiple GPU's includes communication overhead to each device, communication overhead between devices, and even overhead of using a multi-threading framework. As mentioned in section 2.2, DyManD is effective in reducing the communication overhead from host to each of the devices. And the overhead caused by GPUWorker framework is very less. Thus GPUWorker-DyManD implementation causes minimal execution overhead.

3.1 GPUWorker

GPUWorker [7] is a boost threads based multi-GPU framework that was originally part of HOOMD molecular dynamics package. It is a master/slave thread approach where a worker thread holds a CUDA context and the master thread can send messages to many slave threads. The advantages of this framework over other multi-threaded GPU frameworks are:

- A single master thread can call CUDA runtime and kernel functions on multiple GPUs.
- Any CUDA runtime function can be called in the worker thread easily with a simple syntax.
- No performance difference from normal CUDA calls.

But a minor disadvantage is that slight extra latency is added to synchronous calls because of OS thread scheduling. Since this framework has a single master thread that manages work from slave threads bound to different devices, the need for managing different GPU contexts arises. GPUWorker manages different contexts efficiently by binding the context to the corresponding threads. So any usage of GPUWorker functionality within a GPUWorker thread, by default executes within the respective context. In other multi-threaded frameworks different child threads are created from the main thread. Each thread maintains its own context and there is no interference from contexts of other threads. So there is no need of context management for these frameworks.

3.2 DyManD implementation for Multiple GPUs

DyManD algorithm was extended for multi-GPU implementation of Smith-Waterman algorithm which optimizes the communication overhead involved in running the application. The functionality of DyManD was integrated with that of GPUWorker, to get an interface for calling DyManD functionality through GPUWorker. The interface provided is called as GPUWorker-DyManD interface.

DyManD reduces the communication overhead from host to device and GPUWorker reduces overhead caused by multi-threaded frameworks. Thus the overhead involved in running the application on multiple GPUs reduces significantly for the overall application. GPUWorker-DyManD implementation is done in three basic steps: Initialization, Map and Launch. All these steps perform the DyManD based implementation through GPUWorker framework.

- **Initialization**: This step involves creating and binding a GPU context to the current CPU thread. It also involves identification of kernels from the kernel's PTX image and loading them into the current context. The dynamic allocations that follow this initialization allocate memory on the corresponding device and map to an equivalent address in the host's virtual memory.

- **Map**: This is the second important step of the implementation. In this step the data required by the kernel is mapped onto GPU. Data will be in any of the three states: CPU, Shared or GPU. Mapping the data changes its state from CPU to Shared. The Shared state signifies that a specific allocation unit in the data and any other allocation units that are pointed to by this allocation unit recursively in the data should be copied to the GPU before invoking the next GPU function.

- **Launch**: In this step the kernel is not launched immediately onto the device, kernel launch follows transfer of mapped data. Data which is in Shared state is identified and transferred from host to device in terms of basic allocation units. When transferring the data from host to device, read and write permissions for the corresponding pages are removed to prevent false sharing.

After the kernel implementation, when the host tries to access this data, the action results in segmentation fault and the signal handler restores read or write permissions for those pages. The data will be updated only if the transferred data is modified. By transferring the data only when necessary, the dynamic data management ensures mostly acyclic communication. Thus the communication is optimized.

Context management is a major challenge in providing DyManD implementation for multiple GPUs. This is illustrated with an example:

With GPUWorker-DyManD interface, initialization is executed at the start of the main program as shown in the example code below. In the example, gpu0 creates a GPU context on device 0 and binds it to the main thread. When gpu1 GPUWorker thread is created, the current context of main thread gets changed to that of GPU context on device 1.

```
int main ( int argc, char **argv)
{
  GPUWorker gpu0(0);
  gpu0.setModule((char*)imageBytes);

  CUfunction swFunc0;
  gpu0.setCUfunction(&swFunc0,"smithwaterman");

  GPUWorker gpu1(1);
  gpu1.setModule((char*)imageBytes);

  CUfunction swFunc1;
  gpu1.setCUfunction(&swFunc1,"smithwaterman");

    ...
  host_1D_Array_0= malloc(…);

  host_1D_Array_1= malloc(…);
}
```

When the data needs to be allocated on the devices after the initialization step, conflict between GPUWorker's contexts arises. DyManD's initialization binds the GPU context to the current thread's context. Because of multiple contexts created on multiple devices, the main thread must switch between these GPU contexts for ensuring correct implementation. If this is not done the execution fails because of accessing data from an invalid context. Implementing the above code results in runtime error which indicates that the data when tried to transfer to the device 0 without setting its context is an invalid value. The output is shown in the following code.

```
dyncuda.cpp.59 : Invalid value.
Aborted
```

In order to resolve this issue, context management functionality was added to switch between GPU contexts in the GPUWorker-DyManD interface. Context of the current thread can be set to that of GPUWorker thread using the functionality *GPUWorker::setContext*. This function binds the context of the GPUWorker thread to the calling thread. With the context management functionality of the interface, the modified part of the example code is as follows:

```
int main ( int argc, char **argv)
{
    ...
  gpu0.setContext();
  host_1D_Array_0= malloc(…);

  gpu1.setContext();
  host_1D_Array_1= malloc(…);
}
```

By using the GPUWorker's context management functionality and GPUWorker-DyManD interface, correct execution of multi-GPU applications can be ensured. The functionality of GPUWorker-DyManD interface used in this implementation is shown in the Table 1.

Table 1: GPUWorker-DyManD Functionality

Function Prototype	Description
GPUWorker(*dev*)	This creates a context on device with ordinal *dev* and binds it with the corresponding thread.
setModule(*gpuImage*)	Creates a module in the current context and loads kernel image from PTX file.
setCUfunction(*gpuFunc*)	Loads the kernel function from the kernel image into the current module.
setContext()	Binds the GPUWorker thread's context to the calling thread.
getContext()	Returns the context of GPUWorker's thread to the calling thread.
map()	Maps the pointer from host to device and sets it as an argument for the kernel function at particular offset.
ParamSetSize(*gpuFunc*, offset)	Sets the parameter size of the kernel function to be offset value.
launchKernel*(gpuFunc*, dimx, dimy)	Updates the device with the mapped data added to argument list, and launches the kernel function with the specified dimensions.
init()	When a signal is raised, calls a handler for transferring data 2from device to host.

3.3 GPUWorker-DyManD implementation of Smith-Waterman algorithm

As mentioned in section 1.1, Smith-Waterman algorithm is used for finding similarities between nucleotide or protein sequences and return a best match for a given query sequence. The inputs for the algorithm are protein database and query protein arrays and an array containing scores for the sequence match is returned as output. This algorithm can be implemented on multiple GPUs by dividing the input data across multiple devices. When the algorithm uses larger

databases for finding the best match for the input query sequence, multi-GPU implementation is useful.

In GPUWorker-DyManD implementation multiple contexts are created for each of the devices in initialization step. Then input database is divided among corresponding devices and are accessed only in the respective contexts. Database and query arrays of respective contexts are mapped onto the devices. By switching between the GPU contexts, single master thread launches kernels onto different devices and finds a best sequence match on the divided data. Initialization using GPUWorker-DyManD interface is already shown in the example code in section 3.2. Map and Launch steps for GPUWorker-DyManD implementation which includes context management is shown below.

```
int main (int argc, char **argv)
{
  …
  offset = 0;
  gpu0.map(host_1D_Array_0,offset);
  offset += sizeof(host_1D_Array_0);

  gpu0.map(queryArray,offset);
  offset += sizeof(queryArray);

  gpu0.map(query_proteinLength,offset);
  offset+ = sizeof(query_proteinLength);

  gpu0.ParamSetSize(offset);
  gpu0.FuncSetBlockSize(numThreads,1,1);
  gpu0.launchKernel(numBlocks,1);

  gpu1.setContext();

  offset = 0;
  gpu1.map(host_1D_Array_1,offset);
  offset += sizeof(host_1D_Array_1);

 gpu1.map(queryArray_1, offset);
  offset += sizeof(queryArray_1);

  gpu1.ParamSetSize(offset);
  gpu1.FuncSetBlockSize(numThreads,1,1);
  gpu1.launchKernel(numBlocks,1);
  …
}
```

Compared to default multi-GPU implementation, GPUWorker-DyManD implementation gets better performance because it has the advantage of using both GPUWorker and DyManD. The performance gain obtained for this implementation over default implementation is around 3.5x.

4. EXPERIMENTAL RESULTS

In this section the performance of GPUWorker-DyManD implementation of Smith-Waterman algorithm is evaluated on two GPUs. The execution timings of this implementation is compared with default multi-GPU implementation.

The performance baseline is an Intel(R) Xeon(R) node clocked at 2.67GHz with 12 MB of L3 cache. This has 2 sockets and each socket has 6 cores present on it. Overall, it can run 12 threads in parallel. This is a node with Fermi architecture containing two Tesla M2050 GPUs.

In both the implementations, the input database containing protein sequences is divided among multiple devices and sequence matching is done on each of the devices. The optimization of CPU-GPU communication is performed by DyManD functionality called from GPUWorker interface and GPUWorker multi-threading model provides simultaneous execution of kernels on different devices. Both the implementations are run on a node with two GPU devices for different number of input sequences.

The optimizations that are provided by GPUWorker-DyManD implementation over default implementation are: GPUWorker provides concurrent execution of the algorithm on multiple devices. Using this multi-threading model causes slight execution overhead which is less compared to that of other multi-threading models. DyManD provides efficient data management and communication optimization for each GPU. By using DyManD, manual data management which is source of bugs and errors can be avoided. Both GPUWorker and DyManD together give better performance and optimization compared to default implementation.

Execution timings for both the implementations for different number of input sequences are shown in the Table 2. From the results obtained, few observations can be made. For smaller input database sizes, the multi-GPU implementation consists of more of communication and less of computation. So the communication optimization for the input with lesser number of sequences shows greater speed up over normal multi-GPU implementation. As the input array size increases, even the GPU computation increases and the overall execution time is not completely affected by communication. So as the input array size increases, GPUWorker-DyManD interface provides better communication optimization but the speedup achieved is less compared to that of smaller input sizes.

Table 2: Execution timings for both the implementations for different number of input sequences.

Number of Sequences	Execution time for default multi-GPU implementation (in ms)	Execution time for GPUWorker-DyManD multi-GPU implementation (in ms)
10240	3628.652	698.4473
20480	4149.405	1259.717
30720	8064.949	2342.351
40960	8183.823	2558.347
51200	8892.048	3166.526
61440	12219.5	3979.274

The comparison of execution timings for both the implementations is shown in the figure 3. From the graph it is observed that GPUWorker-DyManD implementation of Smith-Waterman algorithm outperforms default implementation when executed on two GPUs for all the input datasets. As dataset size increases, the speedup achieved is consistent across all the input datasets. GPUWorker-DyManD implementation achieves on an average 3.5x performance over default multi-GPU implementation.

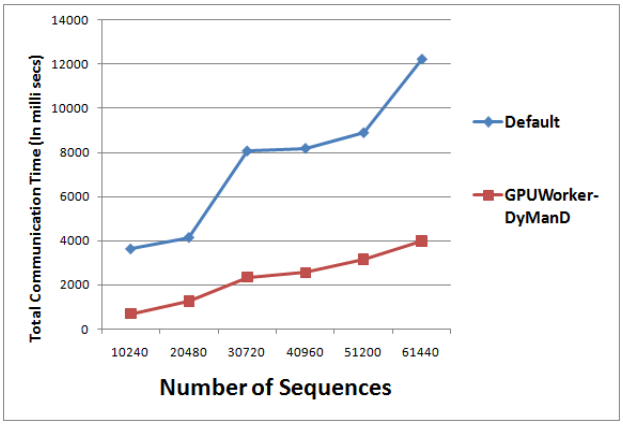

Fig. 3. Smith-Waterman using GPUWorker-DyManD

5. CONCLUSIONS AND FUTURE WORK

Smith-Waterman algorithm is a well-known sequence matching algorithm for finding similarities in a protein database. Several CUDA implementations of this algorithm exist that optimize GPU computation and achieve performance on both single GPU and multiple GPUs. For input datasets of large sizes, multi-GPU implementation provides better performance than single GPU implementation. A multi-GPU implementation was proposed using GPUWorker-DyManD interface for achieving good performance by optimizing communication between CPU and multiple GPUs. GPUWorker multi-threaded framework was used for running the application on multiple GPUs and DyManD provides efficient communication management. So GPUWorker-DyManD interface which consists of both GPUWorker and DyManD optimizes Smith-Waterman algorithm on multiple GPUs. When executed on two GPUs, GPUWorker-DyManD implementation on an average achieves 3.5x performance for all the input data sizes.

As a part of future work, execution of Smith-Waterman can be done on 4 or more GPUs to analyze the scalability of GPUWorker-DyManD implementation. Along with communication optimization between host and devices, device-to-device communication can be utilized and optimized by extending GPUWorker-DyManD functionality.

6. ACKNOWLEDGEMENTS

We offer this work to Bhagawan Sri Sathya Sai Baba, Founder Chancellor of Sri Sathya Sai Institute of Higher Learning. We acknowledge the support and guidance provided to us in the area of CPU-GPU communication optimization by Thomas B. Jablin, a PhD graduate from Princeton University. This work was partially supported by nVIDIA, Pune, grant under Professor Partnership Program and Extreme Science.

7. REFERENCES

[1] D. M. Dang, C. Christara and K. Jackson. GPU pricing of exotic cross-currency interest rate derivatives with a foreign exchange volatility skew model. *SSRN eLibrary*, 2010.

[2] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization. In Mary W. Hall and David A. Padua, editors, *PLDI* , pages 142-151. ACM, 2011.

[3] *NVIDIA Corporation*. CUDA C Best Practices Guide 3.2,2010.

[4] SARJAT SAHNI JUNJIE LI & SANJAY RANKA. Pairwise sequence alignment for very long sequences on gpu.

[5] A. Basumallik and R. Eigenmann. Optimizing irregular shared-memory applications for distributed-memory systems. Number 3, 2006..

[6] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. Dynamically managed data for cpu-gpu architectures. *In Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 165{174, New York, NY, USA, 2012. ACM.

[7] GPUWorker master/ slave multi-GPU approach. https://devtalk.nvidia.com/default/topic/390598/gpuworker-master-slave-multi-gpu-approach/