Study of the implementation of Parsets on GPUs

BTP - II Report

Department of Computer Science and Engineering,
Indian Institute of Technology, Bombay

4th May 2023

Submitted by: Ramyasri Palti
Roll no: 190050078

Abstract

Parallel programming on loosely coupled distributed systems involves many system-dependent tasks such as sensing node availability, creating remote processes, programming inter-process communication and synchronization, etc. Very often, these system-dependent tasks are handled at the programmer level. This has complicated the process of parallel programming on distributed systems. ‘Parset’ is a language construct that captures various kinds of coarse grain parallelism occurring in distributed systems. Parset greatly simplifies writing programs on distributed systems providing transparency to various system-dependent tasks. The initial version of Parset developed was on multi-programmed workstations connected through a local area network. The work includes the parset class, its functions, and its low-level implementation. Our work is to implement Parset on Graphics Processing Unit (GPU) installed system and analyze various aspects such as performance, accessibility, etc. We provide an extensive study of Parset’s motivation, why GPU implementation is introduced, how the GPU brings advantages and challenges, and experiments to observe performance variations and trends using GPU parallelism. We explain how communication between CPU and GPU works on the CUDA platform. We use these observations to implement Parset on a GPU.

Introduction

Parallel programming is an important paradigm in distributed systems. It also has a crucial role in fields such as Machine Learning, where neural networks are trained in batches and in cloud computing as well. Parallel computations significantly speed up repetitive operations that are usually done sequentially. However, distributed systems bring about multiple challenges in the form of synchronization, inter-process communication, node availability, etc. We consider a language construct called ‘Parset’ to reduce the burden on the users in implementing manual CPU resource allocation and synchronization, enabling them to focus on higher-level tasks. Through this project, we present the idea of implementing Parset on GPU using the CUDA library. Using GPUs gives added advantages such as efficient allocation and utilization of GPU resources, versatility using CUDA library functions, reduced latency, etc. By leveraging CUDA's programming model, we can harness the full potential of multiple GPU cores for high-performance computing. But we also need to consider the potential disadvantages, including communication overhead between the CPU and the GPU and the inability to directly limit the number of GPU cores used.

The following report contains all the theory and experiments behind Parsets and GPU implementation. The next section summarises the previous work, and discusses about Parsets. Following that, we describe how GPU is implemented with Parsets, and give results for some experiments depicting the performance.

Previous Work

While implementing parallelism on distributed systems, nodes are created, and each node implements a task. The programmer has to handle node allocation, communication between the nodes, and synchronization once all the parallel programs are executed. The programmer may start his remote processes on heavily loaded nodes, thereby degrading the overall performance of the system. The synchronization and communication implementation has to be written each time programmer wants to run a set of tasks, and it has to be error-prone. To overcome these difficulties, Parset is introduced at the programmer level.

Parsets

Parset is a language construct which captures various kinds of coarse-grain parallelism. It allows the creation of subtasks, locates suitable remote nodes, and gets the code executed on remote nodes. The parset construct consists of a set-type of data structure and a set of functions which operate on this data structure. Parsets are classified into three categories based on the data types of elements in these sets -

- **Simple Parsets**: These contain elements of basic data types and can be used to express SPMD (Single Program Multiple Data) parallelism.
- **Untyped Parsets**: These contain untyped elements and are used in expressing MPMD (Multiple Program Multiple Data) parallelism using polymorphic functions.
- **Function Parsets**: These contain functions as elements and are the most general type of Parsets. They can be used to express MPMD parallelism in a general way.

A Parset is logically ordered based on the order of entry of its elements (first-come-first-served basis). Typed Parsets declare the type of elements they hold. *"parset P of int;"* declares a parset named P that holds elements of the integer type.

The following operations are defined as a part of the Parset object -

- **`insert()`**: Inserts an element into the parset.
- **`flush()`**: Removes all elements from the parset.
- **`get()`**: Retrieves an element from the parset.
- **`delete()`**: Deletes an element from the parset.
- **`getcard()`**: Retrieves the cardinality of the parset.
- **`setcard()`**: Sets the cardinality of the parset.

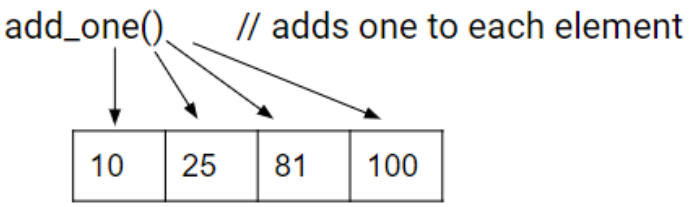
Arguments of functions are tagged as Read-only (RO), Write-only (WO), and Read-Write (RW). This helps us segregate arguments for the parallel execution of the function. It determines how arguments are locked during function execution. For example, if one argument is being written into, then the other arguments are locked and are given only-read access.

Parset also has seq and parallel functions.

- par function call:
  Example: `par process(RO P, RO Q, RO R);`
  The function process0 is applied to each element of P in parallel. If a function has more than one parset as its arguments, then the cardinalities of all of them
  must be the same. This will enable a particular function activation to pick up the corresponding element from each parset.
- seq function call:
  Example: `seq print (RO P);`
  The function print0 is applied to each element of P sequentially in the order of the elements.

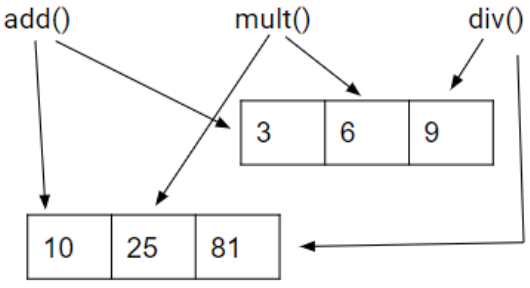
With Parset, programmers can leverage SPMD (Single Program Multiple Data) parallelisms, where multiple instances of the same program are executed in parallel, each operating on different data sets. This form of parallelism is well-suited for tasks that can be divided into independent units, allowing for efficient utilization of resources and enhanced performance.

Ex:



Parset also facilitates MPMD parallelism, where multiple programs execute concurrently, each with its own set of instructions and data. This type of parallelism is valuable in scenarios where different tasks need to be performed simultaneously, offering flexibility and the ability to handle diverse computational requirements.

Ex:



Parsets to GPU

Overview

A Graphics Processing Unit (GPU) is a specialized processor that is designed to handle complex mathematical and computational tasks related to graphics rendering. In the context of distributed systems and parallel computing, GPUs can be used to accelerate the processing of large amounts of data by parallelizing computations across multiple cores. GPUs contain multiple cores that makes them well-suited for multitasking environments, where there are many small, parallelizable tasks that need to be completed simultaneously makes them well-suited for multitasking environments, where there are many small, parallelizable tasks that need to be completed simultaneously.

Additionally, GPUs can take advantage of hardware acceleration for certain types of computations, such as matrix multiplication, which are commonly used in scientific and machine learning applications. By offloading these computations to the GPU, it is possible to achieve significant speedups in the overall execution time of the task.

GPUs also have a high throughput, and due to these properties, they have gained popularity in distributed systems. This has led to the emergence of frameworks like CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language), which enable developers to utilize GPUs for non-graphical computing tasks, such as scientific simulations, machine learning, data analytics, and more.

In our work, we use CUDA to make use of GPUs for general-purpose multi-tasking programs.

Advantages and Disadvantages

GPUs provide the following advantages over CPUs for general tasks -

- Highly parallelizable architecture: Unlike CPUs, which typically have a small number of cores optimized for sequential processing, GPUs have a large number of smaller, more specialized cores that are optimized for parallel processing. These cores are organized into groups known as streaming multiprocessors (SMs), each of which is capable of executing multiple threads simultaneously. When a general-purpose task is submitted to the GPU, it is divided into many smaller tasks, each of which can be executed in parallel across multiple cores. This parallel execution enables the GPU to process the task much faster than a CPU would be able to.
- Cost-effective: GPUs are relatively low-cost devices compared to CPUs, especially when it comes to computing power. With the advent of affordable consumer-level GPUs, it has become easier to harness the power of parallel computing without investing heavily in expensive hardware. A significantly high number of computing devices (1000s of cores) at low cost are made available using GPUs.
- Node Failures: In GPUs, we need not explicitly deal with node failures as the architecture itself handles them. However, in CPUs, when writing programs for distributed systems, the programmer has to verify the code is error-prone and make it robust to run-time errors.
- Memory bandwidth: GPUs have high memory bandwidth, which means that they can move data between their memory and processing units much faster than CPUs. This makes them particularly well-suited to applications that require large amounts of data to be processed

quickly. Due to this, the communication latency is low on GPU than on a network of workstations. GPUs also have a shared memory, which can help to improve system performance by reducing the need for data copies and inter-process communication and enabling processes to operate more efficiently and independently of each other.

While GPUs have several advantages for certain types of computations, they also have some limitations and disadvantages that can make them less suitable for certain types of workloads. Here are some of the disadvantages of GPUs:

- Cannot limit number of cores: Unlike CPUs, which can be scaled up or down based on the requirements of the application, GPUs have a fixed number of processing cores that cannot be easily adjusted. This means that if an application does not require all of the available cores, some of them will be left unused, which can result in wasted resources and reduced efficiency.
- GPU memory is limited: GPUs typically have less memory than CPUs, which can be a limitation for memory-intensive processes. This can be particularly problematic for applications that require large datasets or need to process large amounts of data in real time. While some high-end GPUs have more memory than others, there is still a limit to how much data can be stored and processed at any given time.
- Scheduling in GPU cores is not completely in our hands: While GPUs are designed to execute computations in parallel, the scheduling of tasks across the available cores is typically managed by the GPU driver or hardware. This means that developers have limited control over how tasks are scheduled and executed, which can make it difficult to optimize performance for certain types of workloads.
- Limited compatibility with legacy code: GPUs are designed to work with specialized programming languages and frameworks, such as CUDA and OpenCL. This can make it difficult to use legacy code or libraries that were not designed to work with GPUs, which can limit the applicability of GPU acceleration for certain types of applications.
- High power consumption: GPUs are known for their high power consumption, which can be a concern for data centers and other environments where energy efficiency is important. This can lead to higher operating costs and increased environmental impact, particularly in large-scale deployments.

Weighing the advantages and disadvantages of GPUs in parallelizable tasks, we explore the implementation of Parsets on GPU using CUDA.

CUDA

Overview

NVIDIA's CUDA (Compute Unified Device Architecture) is a platform for parallel computing and programming. Its purpose is to enable the use of GPUs for general-purpose computing tasks. The platform comprises two primary components: the CUDA runtime and the CUDA programming language.

- The CUDA runtime is a software layer that serves as an interface between the CPU (host) and the GPU device. It manages various GPU resources, including initialization, configuration, and coordination. Memory management and data transfer between the CPU and GPU are also handled by the CUDA runtime. The platform's APIs enable developers to allocate and release GPU memory, launch GPU kernels (parallel functions executed on the GPU), and synchronize and transfer data between the CPU and GPU. Additionally, it monitors and reports errors that may occur during GPU operations.
- The CUDA programming language is an extension of the C/C++ programming languages. It provides additional features and syntax to support parallel programming on NVIDIA GPUs. Developers can explicitly specify parallel execution and data movement between the CPU and GPU using this language.

CUDA programming

In CUDA programming, each thread is assigned to run on a single GPU core, allowing for massive parallelism. These threads are organized into groups called thread blocks, where threads within a block can synchronize and communicate through shared memory. Each thread block is assigned to run on a streaming multiprocessor (SM) within the GPU, taking advantage of the SM's computational resources.

To further scale the parallel execution, multiple thread blocks are organized into a grid. Grids consist of numerous thread blocks that can be executed in parallel across multiple streaming multiprocessors in the GPU. This parallel execution across multiple SMs enables efficient utilization of the GPU's processing power and enhances overall performance.

Moreover, each streaming multiprocessor within the GPU has its own cache.The presence of individual caches in each SM allows for efficient data retrieval and reduces the need to access the GPU's global memory, resulting in improved memory access times and overall computational efficiency.

CUDA syntax

Typically, we refer to CPU and GPU system as host and device, respectively.

C program	CUDA

`void c_hello(){`
`  printf("Hello World!\n");`
`}`
 	
`int main() {`
`  c_hello();`
`  return 0;`
`}`
	`__global__ void cuda_hello(){`
	`  printf("Hello World from GPU!\n");`
	`}`
	
	`int main() {`
	`  cuda_hello<<<1,1>>>();`
	`  return 0;`
	`}`

The **`__global__`** specifier indicates a function that runs on device (GPU). These functions can be called by host code for e.g. by “main”. These functions are called **KERNELS**
When a kernel is called, its execution configuration is provided through <<<...>>> syntax, e.g. `cuda_hello<<<1,1>>>()`. In CUDA terminology, this is called "kernel launch"

NVIDIA provides a CUDA compiler called `nvcc` in the CUDA toolkit to compile CUDA code stored with file extension **.cu**

```
nvcc vector_add.cu -o vector_add // to compile code
time ./vector_add                // time taken
nvprof ./vector_add              // to profile the result
```

CPU and GPU have their own memory space. CPU cannot directly access GPU memory, and vice versa. In CUDA terminology, CPU memory is called host memory and GPU memory is called device memory. Pointers to CPU and GPU memory are called host pointer and device pointer, respectively.

For data to be accessible by GPU, it must be presented in the device memory. CUDA provides APIs for allocating device memory and data transfer between host and device memory. Following is the common workflow of CUDA programs.

- Allocate host memory and initialized host data
- Allocate device memory
- Transfer input data from host to device memory
- Execute kernels
- Transfer output from device memory to host

Example program - Vector Addition

```
void main(){
    float *a, *b, *out;
    float *d_a;

    a = (float*)malloc(sizeof(float) * N);

    // Allocate device memory for a
    cudaMalloc((void**)&d_a, sizeof(float) * N);

    // Transfer data from host to device memory
    cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);

    ...
    vector_add<<<1,1>>>(out, d_a, b, N);
    ...

    // Cleanup after kernel execution
    cudaFree(d_a);
    free(a);
}
```

Number of threads used here is 1

CUDA uses a kernel execution configuration <<<...>>> to tell CUDA runtime how many threads to launch on GPU. CUDA organizes threads into a group called "thread block". The kernel can launch multiple thread blocks, organized into a "grid" structure.
<<< M , T >>> - indicates that a kernel launches with a grid of M thread blocks. Each thread block has T parallel threads.

<<1,256>>

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = threadIdx.x;
    int stride = blockDim.x;

    for(int i = index; i < n; i += stride){
        out[i] = a[i] + b[i];
    }
}
```

<<M, 256>> (M is such that there are at least N threads in total)

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    int index = threadIdx.x;
    int stride = blockDim.x;

    for(int i = index; i < n; i += stride){
        out[i] = a[i] + b[i];
    }
}
```

Experiments

Prior to implementing Parset on GPUs, we carried out several experiments aimed at gaining a deeper understanding of the GPU architecture and its behavior. These experiments provided insights into various aspects of GPU programming and performance optimization, allowing us to make informed decisions during the implementation process. The experiments performance trends provide us the motivation to work on implementing parsets on GPUs instead of distributed networks. All experiments are conducted on 8 core intel i5 8th gen processor CPU and 768 core NVIDIA GeForce GTX 1050 Ti GPU

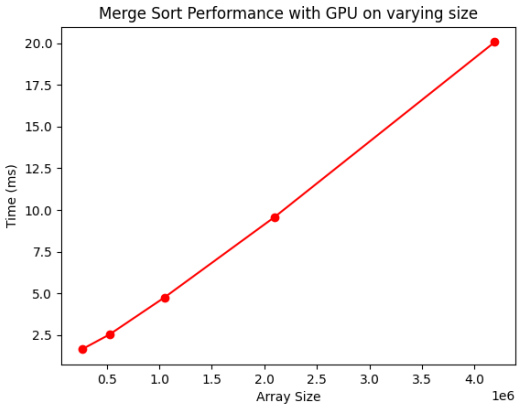
Experiment1

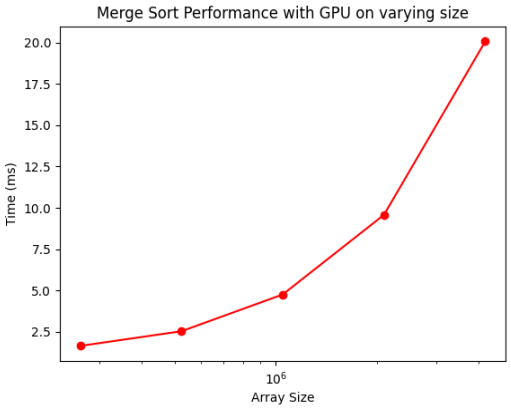
In this experiment, we conducted parallel merge sort implementations on both the CPU and GPU, using varying array sizes, to compare the time taken for sorting. This experiment aimed to evaluate the performance differences between the two platforms and determine the benefits of utilizing the GPU for parallel sorting algorithms.

To ensure a fair comparison, we carefully selected different array sizes, ranging from small to large, to evaluate the scalability of the algorithms on both platforms. We recorded the execution times for sorting each array size on both the CPU and GPU.

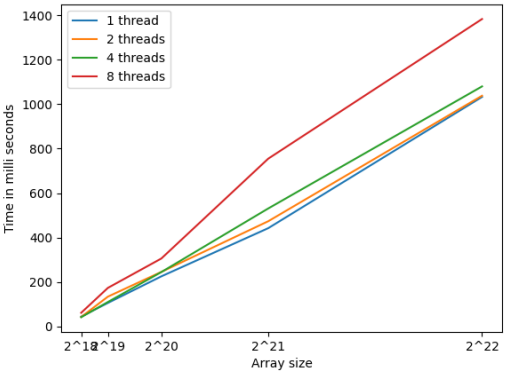
Results

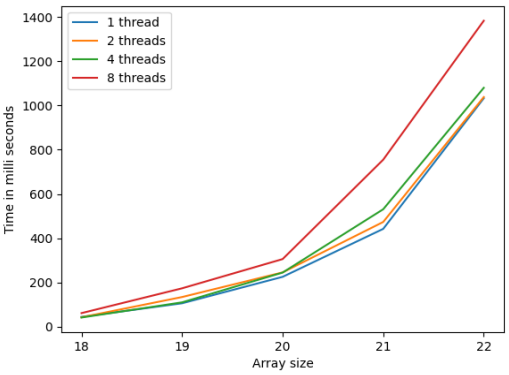
GPU





CPU





Observations

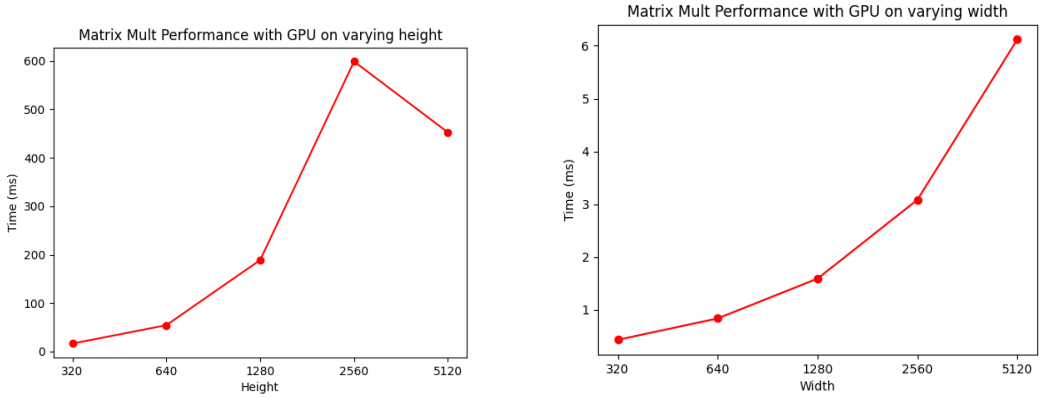
Increasing the number of threads didn't increase the performance, contrary to what we expected. GPU executes Mergesort better by GPU than CPU. However, the growth rates of performance on increasing sizes are similar using both devices.

Experiment 2

In this experiment, we conducted matrix multiplication computations on the GPU using matrices of varying widths and heights. The objective of this experiment was to assess the performance and scalability of the matrix multiplication algorithm on the GPU, while exploring different matrix sizes.

To ensure a fair comparison, we carefully selected different array sizes, ranging from small to large, to evaluate the scalability of the algorithms on both platforms. We recorded the execution times for sorting each array size on both the CPU and GPU.We recorded the execution time of the matrix multiplication algorithm for each combination of matrix dimensions.

Results



Observations

The execution time of the matrix multiplication algorithm is terribly high on the CPU. We observe the growth rate of matrix multiplication is slower than the merge-sort algorithm.

Inference

In the context of GPU computing, processes can generally be categorized into two types: memory intensive processes and compute-intensive processes. Compute-intensive processes, such as matrix multiplications, tend to perform well on GPUs due to their parallel nature and the GPU's ability to execute numerous calculations simultaneously. This is because GPUs are designed to excel at performing highly parallel computations, leveraging their large number of cores.

On the other hand, memory-intensive processes, like merge sort, may not be the best choice to run on GPUs. This is because the overhead of allocating memory on the GPU and transferring data back and forth between the GPU and CPU can outweigh the potential performance gains from parallel execution. The latency and bandwidth limitations associated with data transfer can hinder the overall performance of memory-intensive algorithms on the GPU.

To achieve optimal performance on the GPU, it is crucial to consider the spatial complexity of the algorithm. Spatial complexity refers to the memory requirements and utilization of an algorithm. By optimizing the spatial complexity, one can minimize memory access and data transfer, reducing the potential overhead associated with memory-intensive processes on the GPU. This optimization may involve techniques such as using shared memory efficiently, reducing unnecessary memory allocations, and maximizing data reuse within the GPU's memory hierarchy.

Therefore, when selecting processes to run on the GPU, it is essential to consider the nature of the workload, taking into account whether it is compute-intensive or memory intensive. By optimizing the spatial complexity and carefully choosing the appropriate processes, Parset should harness the full potential of GPU computing and achieve significant performance improvements.

Approaches to the problem

We present two ideas to implement Parsets on GPU

- Parset as a library with class and its member functions defined, along with seq and par functions
- Extend parser and compiler for the new syntax of par and seq calls

Approach 1: Parset as a library

In this approach, parset class, and its member functions are defined in a header file
Parset class has a list of elements, type of element, size of list as member variables. The operations shown above (insert, delete, …) are implemented as member functions. We use template classes and functions to handle unknown elements type.

The seq function can be implemented with ease. But, the par function has to be implemented for SPMD and MPMD differently. For SPMD, a function and the Parsets on which the function has to be implemented are provided.

Ex:

```
void process_spmd(f, A, B, C)
```

This implements `f` using Parsets A, and B and stores the resulting list in parset C. Each parallel task takes an element from A and B and performs f. We assume f is GPU executable parallelised kernel function.
So, our implementation roughly follows the below steps

1. Allocate GPU memory for lists of parsets A, B, and C(say dA, dB, dC)
2. Copy memory into dA and dB from list(A) and list(B)
3. Call f(dA, dB, dC)
4. Copy dC into host memory (say hC)
5. Assign parset C to list hC

For MPMD, a function parset and parsets on which the function parsets are executed are provided.
Ex:

```
void process_mpmd(F, A, B, C)
```

This implements a set of functions written to function parset F, using parsets A and B, and stores the result in parset C
Each parallel task takes one element from function parset F and corresponding elements from A and B. We assume all the functions in F are GPU executable parallised kernel functions
Our implementation roughly follows the below steps

1. Extract an element from F(say f), A(say hA), and B(Say hB)
2. Allocate GPU memory for hA, hB, and C(say dA, dB, dC)
3. Copy memory into dA and dB from hA and hB
4. Assign each f(dA, dB, dC) to a thread and execute all threads in parallel
5. Copy each dC into host memory (say hC)
6. Assign parset C's nth element to result value
7. Synchronize results from all the functions in F

Assigning an f to a thread and synchronizing execution of all tasks is done using CudaStreams.

This kind of implementation poses various problems in coding it.
Firstly, we can not limit the number of parset arguments to process_spmd and process_mpmd, and it can be a variable, which causes difficulty in unpacking parsets, allocating and copying memory for lists in GPU memory.
Secondly, function f as argument uses arguments of type bound to run time, which gives compilation errors.
Thirdly, we are making a lot of assumption on the user code, restricting the number of arguments and expecting the function not to return and copy result to a parset,etc.

```
template <typename T>
void f(T* dA, T* dB, T* dC)
```

These challenges motivate us to see the second approach.

Approach 2: Extended Compiler

Parser and Compiler are extended to ensure the new syntax of par functions. process_spmd and process_mpmd can now take an arbitrary number of parset arguments as the parser handles them.. This approach is left to future work.

Parset rules

E ← par process_spmd(f, S) | par process_mpmd(F, S)

S ← A | B | C | …

S ← AS | BS | CS | …

Future Work

The following ideas can be pursued in the future in GPU implementation of Parsets -

- RO, WO, and RW locks have to be implemented.
- Dynamic load balancing on GPU cores has to be handled as well.
- Optimisation techniques, such as dynamically diverting the load to the CPU based on memory and run-time statistics, have to be implemented too.
- Whole new approach of extending the compiler gives scalability over parset arguments.

Conclusion

The implementation of Parset on GPUs using the CUDA platform offers several advantages over CPU-based Parset implementations. The most significant advantage is the reduced communication latency between the CPU and GPU. Additionally, the GPU architecture allows for a significantly higher number of computing devices, making it a cost-effective solution for parallel processing.

Shared memory on GPUs is another significant advantage, enabling efficient data sharing and communication among parallel threads, which enhances performance. This shared memory is also advantageous because all GPU cores can access it, which means that there is no need to deal with node failures.

However, one drawback of using GPUs for Parset implementation is the limited GPU memory, which can be a limitation for memory-intensive processes. This constraint requires careful management of data transfers between the CPU and GPU, ensuring that only the necessary data is transferred to avoid exceeding GPU memory capacity.

Another disadvantage is the inability to directly limit the number of GPU cores used. This is because the GPU architecture is designed to utilize all available cores for parallel processing. As a result, the programmer must carefully design and optimize their code to make efficient use of GPU resources.

Despite these challenges, the advantages of decreased latency and shared memory outweigh the drawbacks. By addressing the limitations and optimizing the code, we can leverage the immense computational power of GPUs to achieve significant performance improvements in the extended "Parset" implementation using CUDA.

References

1. https://www.nvidia.com/en-us/about-nvidia/webinar-portal/
2. GeForce GTX 1050 Ti | Specifications
3. Joshi, Rushikesh & Ram, D.. (1995). Parset: A language construct for system independent parallel programming on distributed systems. Microprocessing and Microprogramming. 41. 245-259. 10.1016/0165-6074(95)00006-A.
4. https://docs.nvidia.com/cuda/
5. https://www.geeksforgeeks.org/templates-cpp/
6. https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf