Parallel Sorting Algorithms On Hybrid Heterogeneous Servers

Table of Contents

Abstract

Sorting is one of the fundamental algorithms in computer science used in almost any area of area of computing with a large amount of data points or elements. Efficient sorting algorithms are becoming increasingly important as the size and scale of data continues to grow.

Hybrid heterogeneous servers featuring multicore CPUs hosting multiple accelerators (GPUs or FPGAs) dominate the computing landscape due to their ability to efficiently handle very large inputs by performing multiple operations at once.

Because of their importance, efficent sorting algorithms have been the subject of many computer computer science research. Even with these efficient algorithms, sorting large data is still time consuming and benefits from parallel execution. Parallelizing efficient sorting algorithms is challenging and requires efficient design.

Introduction

asdasd

Related Work

Programming Massively Parallel Processors

Architecture Specifications

asasdasd

Multithreaded Cores and Multicore CPUs

Multithreaded and multicore CPUs both exploit concurrency by executing multiple threads, although their design target different objectives. Multithreaded CPUs support concurrent thread execution at the more fine-grained instruction level aiming at better utilizing the resources of CPUs by issuing instructions from multiple threads. Multicore CPUs achieve thread concurrency at a higher level, focusing less on utilization per core and aiming at scalability via replicating cores.[1]

GPUs

The modern GPU has evolved from a fixed function graphics pipeline to programmable parallel processor with computing power exceeding that of multicore CPUs.

With NVIDIA’s Tesla architecture introduced in 2006 in the GeForce 8800 enabled high performance parallel computing applications written in the C programming language using the Computer Unified Device Architecture (CUDA) parallel programming model and development tools. [2]

Cuda Device Architecture

asdasd

Threads, Blocks and Warps

To manage and execute hundreds of threads running several different programs efficiently, GPUs use a process architecture called single instruction multiple threads (SIMT). The SM's SIMT multithreaded instruction unit creates, manages, schedules and executes threads in groups of 32 parallel threads called warps.

Shared Memory

To support computing and C/C++ language needs the Tesla SM implements memory load/store instructions with three read/write memory spaces.
 - local memory: for per thread, private temporary data (implemented in externel DRAM)
 - shared memory: for low latency access to data shared by cooperating threads in the same SM.
- global memory: for data shared by all threads of a computing application (implemented in external DRAM)

Shared memory is located on chip and as a result has a much higher bandwidth and lower latency than local and global memory. Shared memory enables cooperation between threads in a block, when multiple threads in a block use the same data from global memory, shared memory can also be used to access data from global memory only once.

Shared memory is only accessible by threads within the same block, this can become a problem when sorting large inputs that are greater than the size of a block.[3] Currently Tesla GPUs only support block sizes up to 1024.

Tools For Parallel Programming

This section describes the programming models and compilers required for parallel programming.

OpenMP

OpenMP is a parallel programming model for shared memory and distributed shared memory multiprocessors. It is comprised of a set of compiler directives that describe the parallelisms in the source code, along with a supporting library of subroutine available to application.

The directives are instructional notes to any compiler supporting OpenMP. They take the form of source code commens (in Fortran) or #pragma (in C/C++) in order to enhance application portability when porting to non-OpenMP environments.[4]

OpenACC

In contrast to current mainstream GPU programming, such as CUDA and OpenCL, where more explicit compute and data management is necessary, porting of legacy CPU-based applications with OpenACC requires only code annotations without any significant structural changes in the original code, which allows considerable simplification and productivity improvement when hybridizing existing applications.

Programming with OpenACC directives, while greatly simplified is not as flexible as using CUDA or OpenCL. For example, both CUDA and OpenCL provide fine-grained synchronization primitives, such as thread synchronization and atomic operations whereas OpenACC does not.[5]

OpenH

OpenH integrates Pthreads, OpenMP and OpenACC seamlessly to facilitate the development of hybrid parallel programs. An OpenH hybrid parallel program starts as a single main thread, creating a group of Pthreads called hosting Pthreads. A hosting Pthread then leads the execution of software components of the program, either on OpenMP multithreaded component running on the CPU cores or an OpenACC (or OpenMP) component running on one of the accelerators of the server.[6]

CUDA

CUDA is a minimal extension of the C and C++ programming languages. A programmer writes a serial program that calls parallel kernels, which can be simple functions of full programs. The CUDA program executes parallel kernels across a set of parallel threads on the GPU. The programmer organizes these threads into a hierarchy of thread blocks and grids.

The CUDA programming model is similar in style to a single-program multiple data (SPMD) software model – it expresses parallelism explicitly, and each kernel executes on a fixed number of threads. However CUDA is more flexible than most SPMD implementations because each kernel call dynamically creates a new grid with the right number of thread blocks and threads for the application step.

GCC

GNU Compiler Collection is a collection of free and open source compilers from the GNU project. GCC supports OpenMP and OpenACC on NVIDIA GPU's through a tool called nvptx-tools.

NVCC

NVIDIA CUDA Compiler is a proprietary compiler by NVIDIA intended for use with CUDA.

Sorting Algorithms

Sorting algorithms can be classified into stable and unstable algorithms. A stable sorting algorithm preserves the original order of appearance when two elements have equal key values. For example, when sorting the lost [(30, 150), (32, 80), (22, 45), (29, 80)] into a nonincreasing order using income as the key field, a stable sorting algorithm must guarantee that (32, 80) appears before (29, 80) because the former appears before the latter in the original input. An unstable sorting algorithm does not offer such guarantee.[7]

Merging

Merging is a highly important algorithm for many sequential sorting algorithms, most notably the famous merge sort algorithm which we will cover later. But it is especially important in the context of parallel sorting algorithms on GPUs, this is because out input data can be much larger than the size of a single warp meaning that to efficiently make use of shared memory some merging might have to be used.

An ordered merge function takes two sorted lists A and B and merges them into a single sorted list C. We present a parallel ordered merge algorithm in which the input data for each thread is dynamically determined. The dynamic nature of the data access makes it challenging to exploit locality and tiling techniques for improved memory access efficient and performance.[8]

Merge Sort

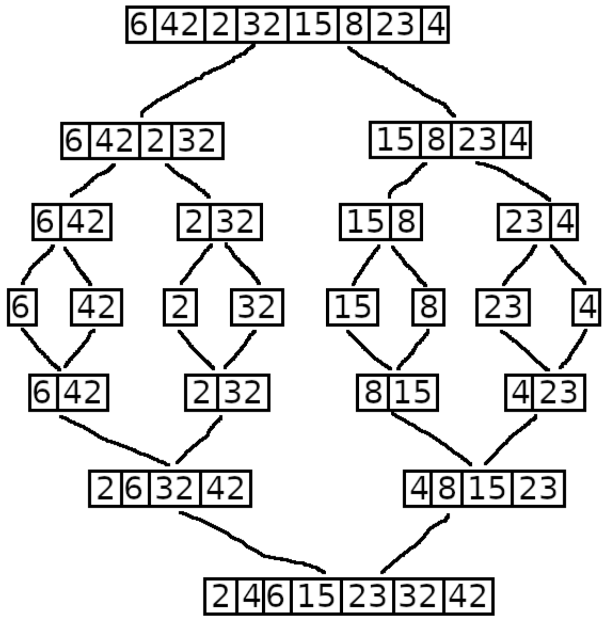
Merge sort sorting algorithm is a very prominent and efficient sorting algorithm. It uses a divide and conquer method for sorting, meaning it breaks the list into smaller sections, sorts these smaller sections and then merges them together to produce the final sorted list.

Intuitively merge sort works by on an array of 'n' elements as given below.

1. If n<1, the entire array is divided into two halves and these arrays are called sub-arrays which have a size "n/2".

2. Apply merge sort on the sub-array

3. The sub-arrays from step 2 are merged into one sorted array.[9]



Radix Sort

One of the sorting algorithm that is highly paralleizeable is radix sort. Radix sort is a non comparison based sorting algorithm that divides the work by distributing the keys that are being sorted into buckets on the basis of a radix value. If the keys consist of multiple digits, the distribution of keys is repeated for each digit until all digits are covered. Each iteration is stable, preserving the order of the keys within each bucket from the previous iteration.

```
__global__ void radix_sort_iter(unsigned int* input, unsigned int* output, unsigned int* bits, unsigned int N, unsigned int iter){
 unisnged int i = blockIdx.x * blockDim.x + threadIdx.x;

 unsigned int key, bit;
 if(i < N){
   key = input[i];
   bit = (key >> iter) & i;
   bits[i] = bit;
 }
 exclusiveScan();
 if(i < N){
   unsigned int numOnesBefore = bits[i];
   unsigned int numOnesTotal = bits[N];
   unsigned int dst = (bit == 0) ? (i - numOnesBefore)
                                  : (N - numOnesTotal – numOnesBefore);
   output[dst] = key;
 }
}
```

Optimizing Using Memory Coalescing

Global memory coalescing is one of the most important optimization techniques in CUDA programs because the number of memory transactions impacts the performance. Therefore, it is important to maximize coalescing by performing optimal memory data layout and the different memory addressing.[10]

Optimizing Using Thread Coarsening

Thread coarsening is an optimization technique that merges multiple threads together reducing the total number of threads while increasing the amount of work performed by each individual thread. The transformation works by replicating instructions that depend on the thread ID by a coarsening factor – for example, with a coarsening factor of 2, each pair of consecutive threads is merged into one.[11]

Measuring Performance

asdasdasd

Conclusions

Asdasdasd

Bibliography

1: A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton and B. Esbaugh, Parallelism via Multithreaded and Multicore CPUs, 2010
2: E. Lindholm, J. Nickolls, S. Oberman and J. Montrym, NVIDIA Tesla: A Unified Graphics and Computing Architecture, 2008
3: Glaskowsky, P.N., NVIDIA’s Fermi: The First CompleteGPU Computing Architecture, 2009
4: Rohit Chandra, Parallel Programmingin OpenMP, 2000
5: T. Hoshino, N. Maruyama, S. Matsuoka and R. Takaki, CUDA vs OpenACC: Performance Case Studies with Kernel Benchmarks and a Memory-Bound CFD Application.,
6: S. Farrelly, R. R. Manumachu and A. Lastovetsky, OpenH: A Novel Programming Model and API for Developing Portable Parallel Programs on Heterogeneous Hybrid Servers, 2024
7: Wen-mei W. Hwu, Izzat El Hajj, Programming Massively Parallel Processors, Chaper 13,
8: Wen-mei W. Hwu, Izzat El Hajj, Programming Massively Parallel Processors, Chaper 12, 2022
9: J. Lobo and S. Kuwelker, Performance Analysis of Merge Sort Algorithms, 2020
10: Dae-Hwan Kim, Evaluation Of The Performance Of GPU Global Memory Coalescing, 2015
11: Alberto Magni, Christophe Dubach, Michael F.P. O’Boyle, A Large-Scale Cross-Architecture Evaluation of Thread-Coarsening, 2013

https://ieeexplore.ieee.org/abstract/document/7054183