Assignment 3 CS4032D Computer Architecture

Team members

S.L. No.	Name	Roll number
1	Pavithra Rajan	B190632CS
2	Cliford Joshy	B190539CS
3	Karthik Sridhar	B190467CS
4	Jesvin Sebastian Madona	B190700CS

File Description

S.L. No.	File	Description
1	matrix_add_cuda.cu	This CUDA C program does Matrix addition
2	matrix_add_openmp.c	This program written in C performs Matrix Addition using OpenMP
3	matrix_add_posix.c	This program written in C performs Matrix Addition using POSIX Threads
4	mult_cuda.cu	This CUDA C program perform Matrix multiplication
5	mult_openmp.c	This C program perform Matrix multiplication with OpenMP
6	mult_posix.c	This C program perform Matrix multiplication with POSIX threads

Q1. Performance comparison of pthread and openMP for a parallelizable problem

A parallelizable problem in computer architecture is a problem that can be divided into smaller independent sub-problems that can be solved concurrently using multiple processors or cores. One such example of a parallelizable problem is Matrix addition.

It is a parallelizable program as it involves adding corresponding elements of two matrices to produce a third matrix. Each element of the output matrix is computed independently of the others, which makes it a good candidate for parallelization.

Parallelizing matrix addition can improve the performance of the program by distributing the workload among multiple processors or cores. Each processor or core can be assigned a subset of the matrix elements to compute, and the results can be combined to produce the final output matrix.

As the size of the input matrices increases, the workload can be distributed among more processors or cores, which can help to maintain performance and reduce the execution time.

Program and Results

In both the cases, we are performing matrix addition of two matrices of size `15000 x 15000`.

Using Pthread

```
gcc matrix_add_posix.c
./a.out
```

```
Execution time without threads: 1.174638 seconds
Execution time with threads: 0.877375 seconds
Speedup: 1.338810
```

Using OpenMP

```
gcc matrix_add_openmp.c -fopenmp
./a.out
```

```
Execution time with threads: 0.250800 seconds
Memory usage: 2.515755 MB
Execution time without threads: 0.797220 seconds
Speedup: 3.178708
```

Analysis

OpenMP and Pthreads are both programming models that can be used to implement parallelism in shared-memory systems. However, OpenMP often provides better speedup than Pthreads for several reasons.

- OpenMP provides a high-level programming interface that is designed to simplify the development of parallel applications. OpenMP allows the programmer to specify parallelism via compiler directives in the codebase, which are automatically translated into threaded code by the compiler. This can make it easier to implement parallelism in a program and can reduce the likelihood of errors.

- OpenMP can automatically manage thread creation and scheduling, which can optimize the use of system resources. OpenMP provides a runtime library that can dynamically allocate threads and distribute workloads among them, which can help to balance the workload and minimize idle time. This can lead to better performance and more significant speedup compared to manually managing threads using Pthreads.

- There is optimization of code at compiler level in the case of OpenMP. The OpenMP compiler can automatically perform optimizations such as loop unrolling and memory prefetching, which can improve cache usage and reduce memory access latency. This can result in faster execution times and more significant speedup compared to hand-written Pthreads code.

Hence, OpenMP often provides better speedup than Pthreads due to its high-level programming interface, automatic thread management, and compiler optimizations. These features can make it easier to implement parallelism in a program, optimize resource usage, and generate more efficient code.

Q2. Compare the performance of pthread, OpenMP and CUDA for two different problems out of which one is purely vectorizable (DLP) and the other is parallelizable (TLP)

A vectorizable program is one that can be optimized to take advantage of vector processing, where a single instruction operates on multiple data elements simultaneously. Vector processing is particularly effective when performing repetitive computations on large amounts of data, as it can reduce the number of instructions required to perform the computation, and thereby improve performance.

Using CUDA to parallelize a purely vectorizable program can result in significant performance improvements. CUDA is a parallel computing platform and programming model that enables developers to harness the power of NVIDIA GPUs (Graphics Processing Units) for general-purpose computing.

CUDA allows the program to be executed on the GPU, which can perform massive amounts of parallel computation in a highly efficient manner. The program can be parallelized using CUDA by dividing the data into small blocks and assigning each block to a separate thread, which can execute in parallel on the GPU.

CUDA also provides support for SIMD operations, allowing multiple data elements to be processed simultaneously, which can further accelerate the computation. Additionally, CUDA provides access to fast shared memory and global memory, which can be used to store and share data between threads.

Overall, using CUDA to parallelize a purely vectorizable program can result in significant speedups, especially on systems with powerful GPUs.

An example of a purely vectorizable program is matrix multiplication. This is because it involves performing a large number of simple, independent arithmetic operations on arrays of data, which can be performed in parallel without any dependencies between the operations.

In matrix multiplication, each element in the resulting matrix is the sum of the products of corresponding elements from the two input matrices. This calculation can be performed independently for each element, making it an ideal candidate for vectorization.

By using vector instructions, such as SIMD operations, the arithmetic operations can be applied to multiple elements simultaneously, further increasing the performance of the computation.

Vectorizable Program

We are performing matrix multiplication of two matrices of size `2000 x 2000`.

Using OpenMP

```
gcc mult_openmp.c -fopenmp
./a.out
```

```
Execution time with OpenMP: 16.765440 seconds
```

Using POSIX Thread

```
gcc mult_posix.c
./a.out
```

```
Execution time with POSIX threads: 46.165331 seconds
```

Using CUDA

```
nvcc mult_cuda.cu
./a.out
```

```
Execution time with CUDA: 0.166911 seconds
```

Analysis

- Comparing the execution time across POSIX threads, OpenMP and CUDA, we can see that there is a significant speed-up in the case of CUDA incorporated matrix multiplication. Matrix multiplication is a computationally intensive task that involves a large number of mathematical operations that can be performed in parallel. GPUs, with their many processing cores, can perform these operations simultaneously, which allows for significant speedups over CPUs.
- In contrast, pthreads and OpenMP are multi-threading libraries that allow us to create threads for concurrent execution of code on a CPU. While these libraries can also provide parallelism, they may not be as efficient as GPUs for matrix multiplication due to the fundamental architectural differences between CPUs and GPUs.
- Hence, the parallel computing architecture of GPUs and the specifically optimized programming model provided by CUDA make it extremely fast for matrix multiplication compared to pthreads and OpenMP.

Parallelizable Program

We take the matrix addition across different sizes and find the execution time.

Analysis

Model	**N = 5**	**N = 50**	**N = 500**	**N = 1000**	**N = 1500**	**N = 2000**	**N = 5000**	**N = 10000**	**N = 15000**
OpenMP	0.000001 s	0.000223 s	0.000440 s	0.004509 s	0.002810 s	0.004599 s	0.025177 s	0.114659 s	0.232911 s
POSIX Threads	0.000002 s	0.001458 s	0.001804 s	0.004058 s	0.008268 s	0.013990 s	0.101341 s	0.395511 s	0.874809 s
CUDA	0.000012 s	0.000012 s	0.000202 s	0.000794 s	0.001840 s	0.003088 s	0.019963 s	0.080502 s	0.072566 s

We can see that in general matrix addition is a bit faster for CUDA approach in contrast to OpenMP and pthread for matrix of smaller sizes but a significant difference can be observed for larger matrices.

Matrix addition with CUDA can be faster than with OpenMP for larger matrix sizes due to the massively parallel architecture of modern GPUs.

CUDA allows for massively parallel execution on the GPU, where thousands of threads can execute simultaneously. This makes it well-suited for computations that can be broken down into many smaller tasks that can be executed in parallel, such as matrix addition.

In contrast, OpenMP uses threads that run on the CPU, which typically has fewer cores than a modern GPU. This means that the parallelization is less fine-grained, and the number of tasks that can be executed in parallel is limited by the number of available cores. This can limit the potential speedup that can be achieved with OpenMP for large matrix sizes.

Additionally, modern GPUs have specialized hardware for performing matrix operations, such as tensor cores, which can further accelerate matrix addition on the GPU.

Therefore, for larger matrix sizes, the massively parallel architecture of the GPU and the specialized hardware for matrix operations can make matrix addition faster with CUDA than with OpenMP.

However, for smaller matrix sizes, the overhead of launching kernels and transferring data between the host and the device in CUDA can outweigh the benefits of parallelization and make OpenMP faster. We can observe this in the case of the matrix size being `5 x 5`.

Installation and Running CUDA programs

- Install the CUDA Toolkit

```
wget https://developer.download.nvidia.com/compute/cuda/repos/ubuntu2204/x86_64/cuda-ubuntu2204.pin
```

```
sudo mv cuda-ubuntu2204.pin /etc/apt/preferences.d/cuda-repository-pin-600
```

```
wget https://developer.download.nvidia.com/compute/cuda/12.1.0/local_installers/cuda-repo-ubuntu2204-12-1-local_12.1.0-530.30.02-1_amd64.deb
```

```
sudo dpkg -i cuda-repo-ubuntu2204-12-1-local_12.1.0-530.30.02-1_amd64.deb
```

```
sudo cp /var/cuda-repo-ubuntu2204-12-1-local/cuda-*-keyring.gpg /usr/share/keyrings/
```

```
sudo apt-get update
```

```
sudo apt-get -y install cuda
```

Once this is done, add the CUDA path variables and reboot your system.

```
export CUDA_HOME="/usr/local/cuda"
export PATH="$PATH:$CUDA_HOME/bin"
```

To check if the nvcc compiler has been installed successfully, run:

```
nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Feb__7_19:32:13_PST_2023
Cuda compilation tools, release 12.1, V12.1.66
Build cuda_12.1.r12.1/compiler.32415258_0
```