The Shortcut Problem

Performance Analysis and Optimisation Project

Cristian Cristea

Faculty of Electronics, Telecommunications and Information Technology
National University of Science and Technology POLITEHNICA Bucharest

June 2024

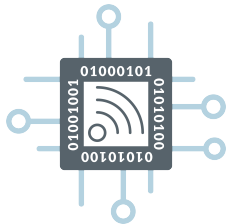

Table of Contents

General Information

Problem Description

We have a directed graph with n nodes. The graph nodes are labelled with numbers $0, 1, \ldots, n-1$. There is a directed edge between each pair of nodes. The cost of the edge between nodes i and j is d_{ij}. We will assume that d_{ij} is a non-negative real number. For convenience, we write $d_{ii} = 0$ for each node i.

However, the costs do not necessarily satisfy the triangle inequality. We might have an edge of cost $d_{ij} = 10$ from node i to j, but there might be an intermediate node k with $d_{ik} = 2$ and $d_{kj} = 3$. Then we can follow the route $i \to k \to j$ at a total cost of $2 + 3 = 5$, while the direct route $i \to j$ would cost 10.

The task is to find for all i and j what is the cost of getting from i to j by taking **at most two edges**. If we write r_{ij} for the result, then

$$r_{ij} = \min_k (d_{ik} + d_{kj}) \,,$$

where k ranges over $0, 1, \ldots, n-1$. Note that routes such as $i \to i \to j$ will also be considered, hence a path of one edge will be found if it happens to be the cheapest.

Problem Description

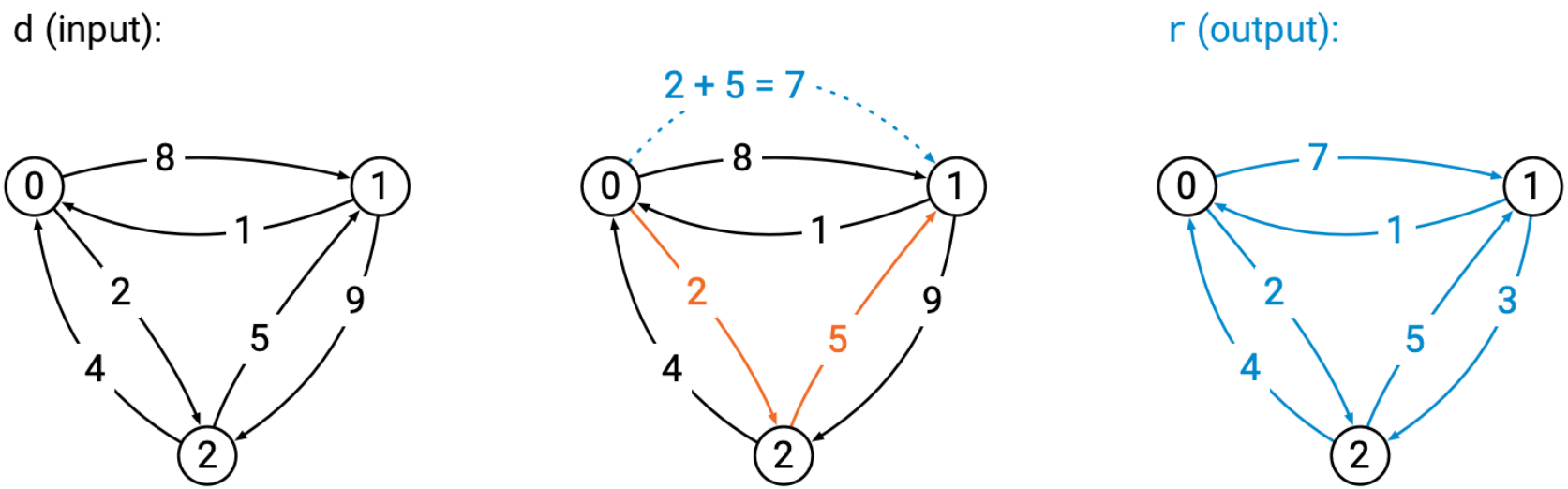

Figure 1: Graph Example

Progress

Straightforward (naive) implementation

```cpp
template <typename T>
auto Shortcut(Graph<T> const & graph) -> Graph<T>
{
    Graph<T> result{graph.size()};

    for (std::size_t row = 0; row < graph.size(); ++row)
    {
        for (std::size_t col = 0; col < graph.size(); ++col)
        {
            T minimum{max_value<T>::get()};

            for (std::size_t k = 0; k < graph.size(); ++k)
            {
                minimum = std::min(minimum, graph[row, k] + graph[k, col]);
            }

            result[row, col] = minimum;
        }
    }

    return result;
}
```

Code Snippet 1: Naive C++ implementation

Implementations

Implementations

- **Naive** – The straightforward approach which establishes the baseline for benchmarking the other implementations.

Implementations

- **Naive** – The straightforward approach which establishes the baseline for benchmarking the other implementations.
- **NaiveOpenMP** – The improved baseline approach which just distributes the work across all threads using `OpenMP`.

Implementations

- **Naive** – The straightforward approach which establishes the baseline for benchmarking the other implementations.
- **NaiveOpenMP** – The improved baseline approach which just distributes the work across all threads using `OpenMP`.
- **Cached** – The matrix of the graph is accessed in a cache-friendly manner by rows but inefficiently by columns. To address this issue, the matrix is transposed, ensuring that both row and column accesses are optimised for cache efficiency.

Implementations

- **Naive** – The straightforward approach which establishes the baseline for benchmarking the other implementations.
- **NaiveOpenMP** – The improved baseline approach which just distributes the work across all threads using `OpenMP`.
- **Cached** – The matrix of the graph is accessed in a cache-friendly manner by rows but inefficiently by columns. To address this issue, the matrix is transposed, ensuring that both row and column accesses are optimised for cache efficiency.
- **CachedOpenMP** – The Cached implementation which just distributes the work across all threads using `OpenMP`.

Implementations

- **Naive** – The straightforward approach which establishes the baseline for benchmarking the other implementations.
- **NaiveOpenMP** – The improved baseline approach which just distributes the work across all threads using `OpenMP`.
- **Cached** – The matrix of the graph is accessed in a cache-friendly manner by rows but inefficiently by columns. To address this issue, the matrix is transposed, ensuring that both row and column accesses are optimised for cache efficiency.
- **CachedOpenMP** – The Cached implementation which just distributes the work across all threads using `OpenMP`.
- **SIMD** – To leverage the CPU's SIMD instructions, the matrix elements of the graph are stored as vector types and padded as necessary. Additionally, the transposed approach is employed to optimise cache access.

Implementations

- **Naive** – The straightforward approach which establishes the baseline for benchmarking the other implementations.
- **NaiveOpenMP** – The improved baseline approach which just distributes the work across all threads using `OpenMP`.
- **Cached** – The matrix of the graph is accessed in a cache-friendly manner by rows but inefficiently by columns. To address this issue, the matrix is transposed, ensuring that both row and column accesses are optimised for cache efficiency.
- **CachedOpenMP** – The Cached implementation which just distributes the work across all threads using `OpenMP`.
- **SIMD** – To leverage the CPU's SIMD instructions, the matrix elements of the graph are stored as vector types and padded as necessary. Additionally, the transposed approach is employed to optimise cache access.
- **SIMDOpenMP** – The SIMD implementation which just distributes the work across all threads using `OpenMP`.

Implementations

- **Naive** – The straightforward approach which establishes the baseline for benchmarking the other implementations.
- **NaiveOpenMP** – The improved baseline approach which just distributes the work across all threads using `OpenMP`.
- **Cached** – The matrix of the graph is accessed in a cache-friendly manner by rows but inefficiently by columns. To address this issue, the matrix is transposed, ensuring that both row and column accesses are optimised for cache efficiency.
- **CachedOpenMP** – The Cached implementation which just distributes the work across all threads using `OpenMP`.
- **SIMD** – To leverage the CPU's SIMD instructions, the matrix elements of the graph are stored as vector types and padded as necessary. Additionally, the transposed approach is employed to optimise cache access.
- **SIMDOpenMP** – The SIMD implementation which just distributes the work across all threads using `OpenMP`.
- **OpenCL** – The straightforward approach was implemented as an `OpenCL` kernel.

Implementations

- **Naive** – The straightforward approach which establishes the baseline for benchmarking the other implementations.
- **NaiveOpenMP** – The improved baseline approach which just distributes the work across all threads using `OpenMP`.
- **Cached** – The matrix of the graph is accessed in a cache-friendly manner by rows but inefficiently by columns. To address this issue, the matrix is transposed, ensuring that both row and column accesses are optimised for cache efficiency.
- **CachedOpenMP** – The Cached implementation which just distributes the work across all threads using `OpenMP`.
- **SIMD** – To leverage the CPU's SIMD instructions, the matrix elements of the graph are stored as vector types and padded as necessary. Additionally, the transposed approach is employed to optimise cache access.
- **SIMDOpenMP** – The SIMD implementation which just distributes the work across all threads using `OpenMP`.
- **OpenCL** – The straightforward approach was implemented as an `OpenCL` kernel.
- **OpenCLSwapped** – The same straightforward `OpenCL` kernel was used, but with the row and column indices swapped to enhance cache access efficiency.

Results

Results

Implementation	$N = 100$	$N = 200$	$N = 500$	$N = 1000$	$N = 2000$	$N = 5000$
Naive	0 ms	5 ms	70 ms	1 065 ms	40 961 ms	974 426 ms
NaiveOpenMP	20 ms	20 ms	45 ms	315 ms	8 667 ms	172 436 ms
Cached	0 ms	2 ms	33 ms	265 ms	2 888 ms	41 513 ms
CachedOpenMP	30 ms	30 ms	30 ms	71 ms	672 ms	13 703 ms
SIMD	0 ms	5 ms	50 ms	317 ms	3 213 ms	42 535 ms
SIMDOpenMP	30 ms	30 ms	40 ms	127 ms	596 ms	9 735 ms
OpenCL	0 ms	0 ms	6 ms	59 ms	1 823 ms	40 173 ms
OpenCLSwapped	0 ms	0 ms	1 ms	31 ms	290 ms	4 672 ms
SoA	0 ms	0 ms	0 ms	7 ms	271 ms	6 453 ms
SoAOpenMP	20 ms	20 ms	20 ms	22 ms	77 ms	1 162 ms
SoAOpenCL	0 ms	0 ms	0 ms	1 ms	8 ms	122 ms

Table 1: Results comparison for different input sizes

Conclusions

Conclusions

- **Distributing Computation Load** – Utilising APIs like `OpenMP` to distribute the computation load across multiple threads significantly increases speed. However, this approach introduces some latency due to thread spawning and context switching times.

Conclusions

- **Distributing Computation Load** – Utilising APIs like `OpenMP` to distribute the computation load across multiple threads significantly increases speed. However, this approach introduces some latency due to thread spawning and context switching times.

- **Optimising Memory Access** – Maintaining a high rate of cache hits is crucial for performance optimisation since memory access is often the primary bottleneck in computational processes.

Conclusions

- **Distributing Computation Load** – Utilising APIs like `OpenMP` to distribute the computation load across multiple threads significantly increases speed. However, this approach introduces some latency due to thread spawning and context switching times.
- **Optimising Memory Access** – Maintaining a high rate of cache hits is crucial for performance optimisation since memory access is often the primary bottleneck in computational processes.
- **Leveraging CPU SIMD Instructions** – Exploiting all CPU resources, including SIMD instruction extensions, enhances performance by taking advantage of parallel processing capabilities inherent in these instructions.

Conclusions

- **Distributing Computation Load** – Utilising APIs like `OpenMP` to distribute the computation load across multiple threads significantly increases speed. However, this approach introduces some latency due to thread spawning and context switching times.
- **Optimising Memory Access** – Maintaining a high rate of cache hits is crucial for performance optimisation since memory access is often the primary bottleneck in computational processes.
- **Leveraging CPU SIMD Instructions** – Exploiting all CPU resources, including SIMD instruction extensions, enhances performance by taking advantage of parallel processing capabilities inherent in these instructions.
- **Utilising Specialised Hardware** – Delegating the processing of parallel programs to specialised hardware such as GPUs significantly reduces computation time due to the massive parallelism these devices offer.

Conclusions

- **Distributing Computation Load** – Utilising APIs like `OpenMP` to distribute the computation load across multiple threads significantly increases speed. However, this approach introduces some latency due to thread spawning and context switching times.
- **Optimising Memory Access** – Maintaining a high rate of cache hits is crucial for performance optimisation since memory access is often the primary bottleneck in computational processes.
- **Leveraging CPU SIMD Instructions** – Exploiting all CPU resources, including SIMD instruction extensions, enhances performance by taking advantage of parallel processing capabilities inherent in these instructions.
- **Utilising Specialised Hardware** – Delegating the processing of parallel programs to specialised hardware such as GPUs significantly reduces computation time due to the massive parallelism these devices offer.
- **Advanced Performance Techniques** – Achieving state-of-the-art performance requires employing advanced techniques. For CPUs, this includes strategies like register reuse and prefetching. For GPUs, optimising data reuse in shared memory is essential for maximising performance.

Thank you for your attention! Questions?