گزارش آزمایش بررسی عملکرد جمع آرایه‌ها

نام درس: پردازش موازی

عنوان آزمایش: مقایسه عملکرد جمع دو بردار در سه روش متفاوت

اعضای گروه: پارسا مبینی دهکردی، محمد صدرا افشار

استاد درس: دکتر ابراهیم زارعی

۱. هدف آزمایش

هدف از این آزمایش، بررسی و مقایسه عملکرد سه روش مختلف برای جمع دو بردار است:

- جمع آرایه‌ها به صورت سنتی [1]
- جمع با استفاده از بردارها [2]
- جمع با استفاده از بردارهای بهینه‌شده[3]

این مقایسه به منظور درک تأثیر ساختار داده و بهینه‌سازی‌های کامپایلر بر روی عملکرد برنامه انجام می‌شود.

۲. روش‌های پیاده‌سازی

الف) آرایه

- استفاده از آرایه‌های استاتیک
- دسترسی مستقیم به حافظه
- عدم سربار[4] مدیریت حافظه
- کنترل کامل بر اختصاص[5]

ب) بردار [6]

- استفاده از ساختار داده std::vector در C++

[1] Array-based
[2] Vector-based
[3] Optimized Vector-based
[4] Overhead
[5] Allocation
[6] Vector

• مدیریت خودکار حافظه

• سربار اندک برای مدیریت حافظه

• امکانات اضافی مانند resize و push_back

ج) بردار بهینه‌شده

• استفاده از بهینه‌سازی‌های کامپایلر

• استفاده از دستورات تک دستور - چند داده [7]

• بهره‌گیری از بردارسازی[8]

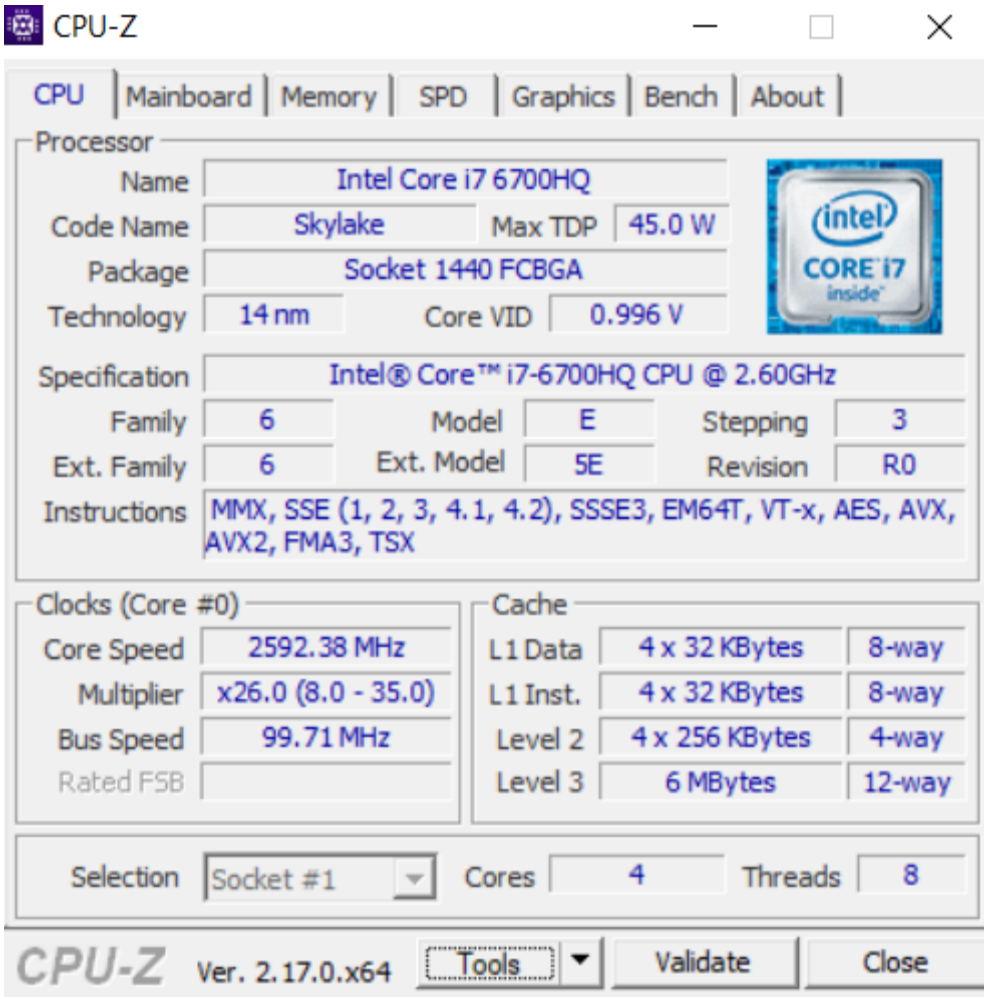
• استفاده از pragma directives یا compiler flags مانند -O3

۳. مشخصات سخت‌افزاری محیط آزمایش

بر اساس اطلاعات گزارش‌شده توسط نرم‌افزار CPU-Z، سیستم استفاده شده از پردازنده Intel Core i7-6700HQ از نسل اسکای‌لیک (Skylake) با فرآیند ساخت ۱۴ نانومتری استفاده می‌کند. این پردازنده دارای ۴ هسته فیزیکی و ۸ رشته پردازشی (با قابلیت Hyper-Threading) بوده و با سرعت پایه ۲.۶ گیگاهرتز (و حداکثر ضریب ۳۵) عمل می‌کند. پردازنده مجهز به حافظه کش سطوح ۱، ۲ و ۳ به ترتیب با ظرفیت‌های ۱۲۸ کیلوبایت، ۱ مگابایت و ۶ مگابایت است و از مجموعه دستورات پیشرفته‌ای مانند AVX2 و FMA3 پشتیبانی می‌کند. این CPU با TDP برابر ۴۵ وات، عمدتاً در لپ‌تاپ‌ها و سیستم‌های همه‌کاره با کارایی بالا به کار می‌رود.

[7] SIMD (Single Instruction, Multiple Data)

[8] Vectorization



CPU specification reported by CPU-Z

۴. مشخصات نرم‌افزاری محیط آزمایش

Programming Language	C++
Compiler	GCC 11.3.0
IDE	Visual Studio Code
Flags for Array	-O0
Flags for Vector	-O2
Flags for Optimized Vector	-O3 -march=native -ftree-vectorize

۵. پیاده‌سازی

روش ۱: جمع با آرایه

```
void arrayAdd(int* A, int* B, int* C, int n) {
    for (int i = 0; i < n; i++) {
        C[i] = A[i] + B[i];
    }
}
```

روش ۲: جمع با بردار

```
void vectorAdd(vector<int>& A, vector<int>& B, vector<int>& C) {
    for (size_t i = 0; i < A.size(); i++) {
        C[i] = A[i] + B[i];
    }
}
```

روش ۳: جمع با بردار بهینه‌شده

```
#include <immintrin.h>
 int i = 0;
 for (; i + 8 <= Bound; i += 8) {

     __m256i va = _mm256_load_si256((__m256i*) & Av[i]);
     __m256i vb = _mm256_load_si256((__m256i*) & Bv[i]);

     __m256i vc = _mm256_add_epi32(va, vb);

     _mm256_store_si256((__m256i*) & Cv[i], vc);
 }

 for (; i < Bound; ++i) {
     Cv[i] = Av[i] + Bv[i];
 }
```

اندازه‌گیری زمان: برای اندازه‌گیری دقیق زمان اجرا از کتابخانه chrono استفاده شد.

```
#include <chrono>
using namespace std::chrono;

auto start = high_resolution_clock::now();
// execute the function
auto end = high_resolution_clock::now();
auto duration = duration_cast<microseconds>(end - start);
```

پارامترهای آزمایش

- اندازه آرایه‌ها: $10^2, 10^3, 10^4, 10^5, 10^6, 10^7$
- تعداد تکرار: پنجاه بار برای هر اندازه (محاسبه میانگین)
- مقادیر ورودی: اعداد تصادفی بین صفر تا نود و نه
- شرایط اجرا: حداقل بار سیستم، بدون برنامه‌های پس‌زمینه سنگین

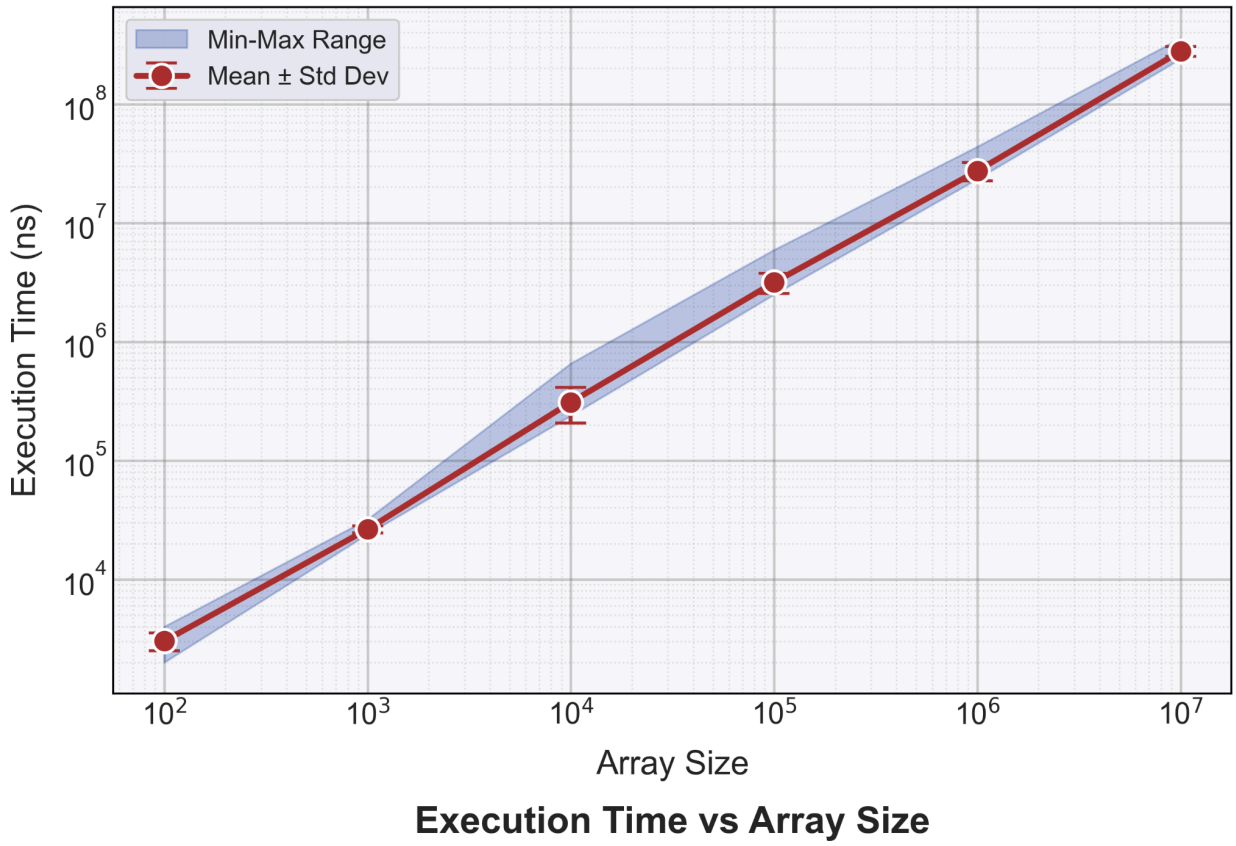
۶. نتایج آزمایش

۶.۱. نمودار خطی پیاده سازی با آرایه

نمودار زیر زمان اجرا (برحسب نانوثانیه) را در مقابل اندازه آرایه (با مقیاس لگاریتمی) برای روش جمع با آرایه سنتی (با `O0`) نشان می دهد.

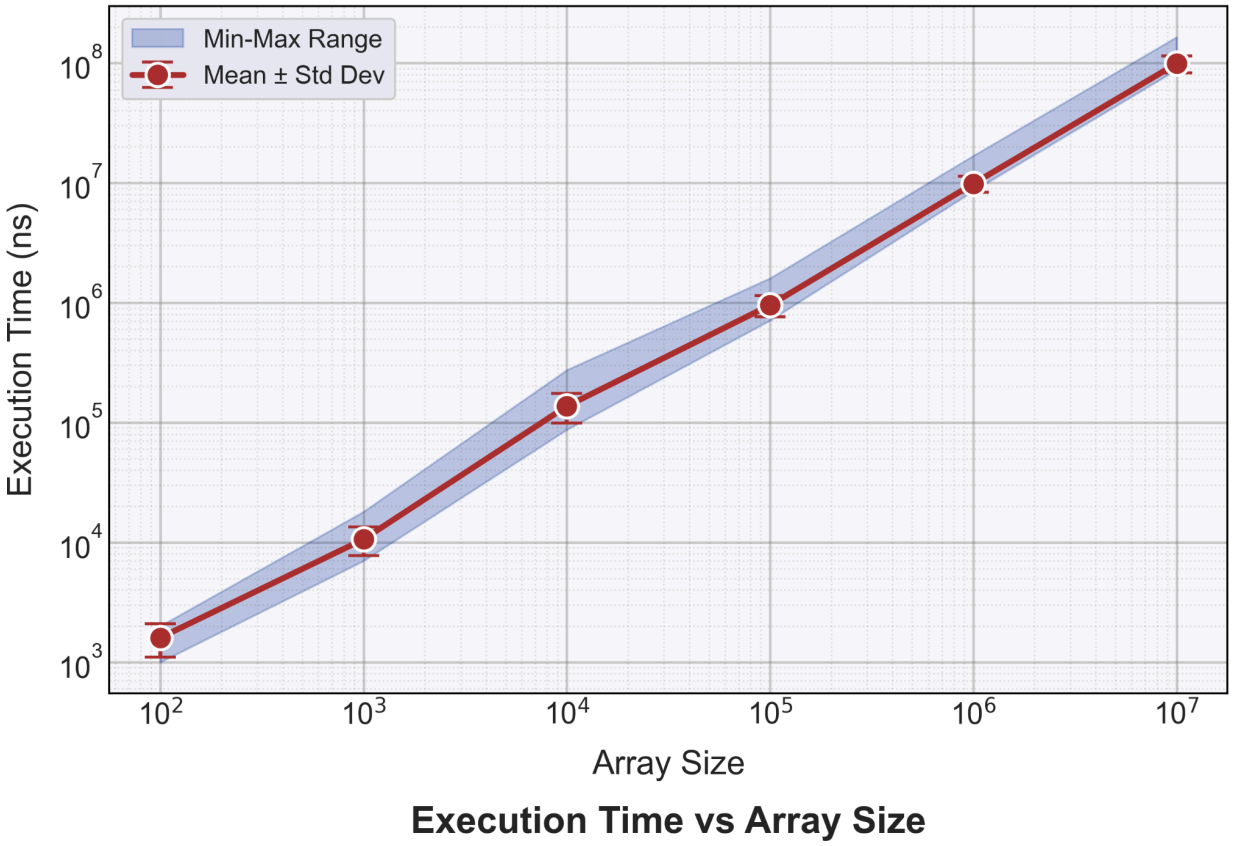
۶.۲. نمودار خطی پیاده سازی با بردار

این نمودار، میانگین زمان اجرا (بر حسب نانوثانیه) را در مقابل اندازه آرایه (با مقیاس لگاریتمی) برای روش جمع با استفاده از `std::vector` (با `-O2`) نشان می‌دهد.



۶.۲. نمودار خطی پیاده سازی با بردار

این نمودار، میانگین زمان اجرا (بر حسب نانوثانیه) را در مقابل اندازه آرایه (با مقیاس لگاریتمی) برای روش جمع با استفاده از `std::vector` (با `-O2`) نشان می‌دهد.

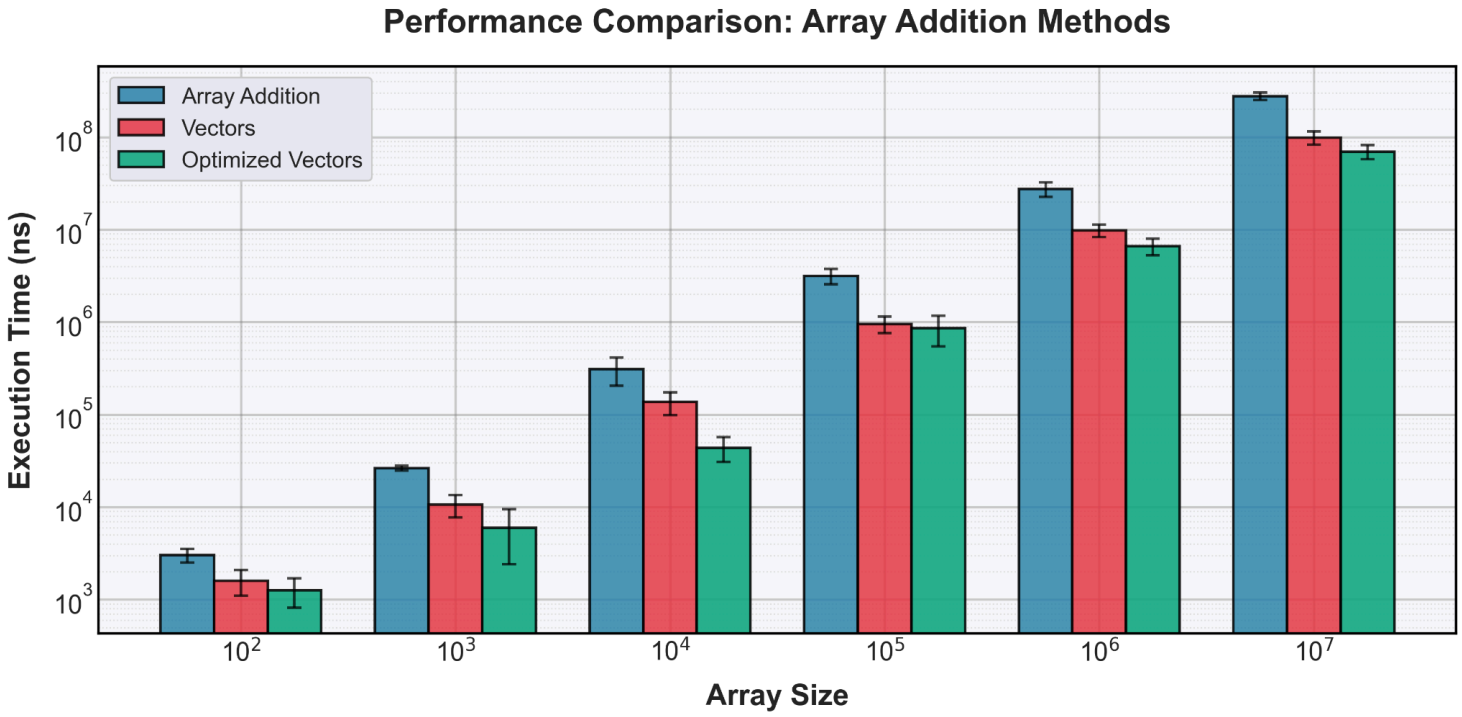


۶.۳. نمودار خطی پیاده سازی با بردار بهینه شده

این نمودار، میانگین زمان اجرا (بر حسب نانوثانیه) را در مقابل اندازه آرایه (با مقیاس لگاریتمی) برای روش جمع با بردار بهینه‌شده (با -O3 و تک دستور-چند داده) نشان می‌دهد.

۶.۴. مقایسه نتایج (میانگین زمان اجرا)

این نمودار میله‌ای، **میانگین زمان اجرا** (بر حسب نانو ثانیه و در مقیاس لگاریتمی) سه روش جمع آرایه‌ها (آرایه، بردار، بردار بهینه‌شده) را در شش اندازه مختلف آرایه (از 10^2 تا 10^7) مقایسه می‌کند.



جدول زیر، مقادیر عددی **میانگین زمان اجرای** به‌دست‌آمده برای هر سه روش را در شش اندازه مختلف آرایه، برای تحلیل دقیق‌تر، ارائه می‌کند.

Performance Comparison: Array Addition Methods			
Array Size	**Array (ns)**	**Vectors (ns)**	**Optimized Vectors(ns)**
100	3,020	1,600	1,260
1,000	26,400	10,600	5,960
10,000	309,180	136,740	43,720
100,000	3,155,600	953,080	859,220
1,000,000	27,398,420	9,819,620	6,649,360
10,000,000	277,949,760	98,814,480	69,597,400

۶.۵ آنالیز آماری

جدول آماری: نتایج عملکرد هر سه روش جمع آرایه‌ها (Array, Vector, Optimized) را بر حسب نانو ثانیه در ۶ اندازه مختلف داده نشان می‌دهد.

Array Size	Metric	Array (O0)	Vector (O2)	Optimized Vector (O3 + SIMD)
10^2	Mean	3020	1600	1260
	Std Dev	515	495	443
	Min	2000	1000	1000
	Max	4000	2000	2000
10^3	Mean	26400	10600	5960
	Std Dev	1726	2864	3557
	Min	24000	7000	2000
	Max	32000	18000	15000
10^4	Mean	309180	136740	43720
	Std Dev	103164	37737	13096
	Min	238000	87000	26000
	Max	657000	274000	86000
10^5	Mean	3155600	953080	859220
	Std Dev	604142	191399	312978
	Min	2494000	710000	497000
	Max	5931000	1597000	1997000
10^6	Mean	27398420	9819620	6649360
	Std Dev	4868984	1499646	1362326
	Min	23393000	8546000	5827000
	Max	43847000	16807000	12561000
10^7	Mean	277949760	98814480	69597400
	Std Dev	26984938	15892590	12153405
	Min	240219000	87737000	60804000
	Max	359034000	164097000	119547000

۷. تحلیل و بحث

نتایج آزمایش، که در نمودار مقایسه عملکرد (بخش ۶.۴) و جدول آماری (بخش ۶.۵) ارائه شده‌اند، به‌طور واضح تفاوت عملکردی سه روش پیاده‌سازی جمع آرایه‌ها را نشان می‌دهند.

۷.۱. مقایسه عملکرد کلی

با افزایش اندازه آرایه از 10^2 تا 10^7، سه روش به ترتیب زیر از لحاظ سرعت، از سریع‌ترین تا کندترین، قرار می‌گیرند:

1. **بردار بهینه‌شده**
2. **بردار**
3. **آرایه سنتی**

این روند نشان می‌دهد که:

- استفاده از ساختار std::vector به تنهایی (روش ۲) عملکردی به مراتب بهتر از پیاده‌سازی سنتی با آرایه (روش ۱) داشته است.
- استفاده از بهینه‌سازی‌های پیشرفته کامپایلر (روش ۳) به‌طور چشمگیری از هر دو روش دیگر سریع‌تر است.

۷.۲. تحلیل عملکرد آرایه با پرچم O0

پیاده‌سازی با آرایه سنتی با پرچم -O0 (بدون بهینه‌سازی) کامپایل شده است. این پرچم باعث می‌شود کامپایلر کمترین تغییرات را در کد اعمال کند، و در نتیجه، عملکرد بسیار کندتر است. دلیل اصلی کندی، عدم استفاده از بهینه‌سازی‌هایی مانند:

- **بهره‌گیری از کش** [9] **:** عدم استفاده موثر از حافظه نهان (کش) پردازنده.
- **بردارسازی یا تک دستور - چند داده:** عدم تبدیل حلقه‌ی ساده‌ی جمع به دستورات پردازش موازی داده‌ها که می‌تواند چندین عمل جمع را همزمان انجام دهد.

به‌عنوان مثال، برای آرایه‌ای به اندازه 10^7، روش آرایه سنتی به صورت میانگین 277,949,760 نانوثانیه زمان برده است.

[9] Cache

۷.۳. تحلیل عملکرد بردار با پرچم O2

پیاده‌سازی با std::vector با پرچم -O2 (بهینه‌سازی متوسط) کامپایل شده است. با وجود سربار اندک مدیریت حافظه خودکار، پرچم -O2 به کامپایلر اجازه داده است تا بهینه‌سازی‌هایی مانند طوقه ی بی چرخش[10]، بهینه‌سازی دسترسی به حافظه و احتمالاً برخی بردارسازی های پایه را اعمال کند.

این بهینه‌سازی‌ها باعث شده است که عملکرد بردار به مراتب بهتر از آرایه سنتی باشد. برای اندازه 10^7، زمان اجرای روش بردار 98,814,480 نانوثانیه بوده که تقریباً بدون سربار ۸ و به طور میانگین به نسبت ۲.۸ برابر سریع‌تر از روش آرایه سنتی است.

۷.۴. تحلیل عملکرد بردار بهینه‌شده

این روش با پیشرفته‌ترین پرچم‌ها (-O3 -march=native -ftree-vectorize) و دستور صریح #pragma omp simd کامپایل شده است.

- **O3:** شدیدترین سطح بهینه‌سازی را اعمال می‌کند.
- **بردارسازی:** پرچم -ftree-vectorize و #pragma omp simd کامپایلر را مجبور می‌کند تا از قابلیت‌های پردازنده (مانند AVX2 در پردازنده i7 6700HQ) استفاده کند. این به معنی انجام چندین عملیات جمع (۸ جمع) تنها با یک دستورالعمل است، که مصداق بارز **پردازش موازی داده‌ها** است.

به دلیل تک دستور- چند داده، عملکرد این روش به شدت بهبود یافته است. برای اندازه 10^7، زمان اجرای آن 6.95974×10^7 نانوثانیه بوده که تقریباً **۴ برابر سریع‌تر** از روش آرایه سنتی و **۱.۴ برابر سریع‌تر** از روش بردار ساده است.

۷.۵. تأثیر مقیاس‌پذیری [11]

- در **اندازه‌های کوچک** (مانند 10^2 و 10^3)، تفاوت زمان‌های اجرا نسبتاً کم است. سربار فراخوانی تابع و مدیریت حافظه در روش‌های برداری ممکن است مزایای بهینه‌سازی را تحت‌الشعاع قرار دهد.
    - در 10^2: زمان اجرای الگوریتم به‌ترتیب برای پیاده‌سازی مبتنی بر آرایه، بردار، و بردار بهینه‌شده به میزان 3020 نانوثانیه، 1600 نانوثانیه، و 1260 نانوثانیه اندازه‌گیری شد.

[10] Loop Unrolling

[11] Scalability

- در **اندازه‌های بزرگ** (مانند 10^6 و 10^7)، تأثیر بهینه‌سازی‌ها، به‌ویژه بردارسازی، به شدت افزایش می‌یابد. در این حالت، زمان صرف‌شده برای محاسبات بر سربار غلبه می‌کند و مزیت تک دستور - چند داده به‌طور کامل آشکار می‌شود، که نشان‌دهنده **مقیاس‌پذیری بهتر** این روش‌ها برای حجم بالای داده است.

۸. نتیجه‌گیری

۸.۱. خلاصه یافته‌ها

آزمایش مقایسه عملکرد سه روش جمع آرایه‌ها نشان داد که **بهینه‌سازی‌های کامپایلر** و **ساختار داده** تأثیر مستقیم و چشمگیری بر سرعت اجرای برنامه‌هایی با ماهیت پردازش موازی داده‌ها[12] دارند.

- **روش آرایه با O0:** کندترین عملکرد را به دلیل عدم استفاده از بهینه‌سازی‌های کامپایلر نشان داد.
- **روش بردار با O2:** عملکرد بسیار بهتری نسبت به آرایه سنتی داشت که عمدتاً به‌خاطر فعال شدن بهینه‌سازی‌های اولیه کامپایلر بود.
- **روش بردار بهینه‌شده O3 و تک دستور - چند داده:** سریع‌ترین روش بود. استفاده از پرچم‌های `-O3` و دستور `#pragma omp simd` توانست قابلیت‌های **تک دستور - چند داده** پردازنده را فعال کند تا چندین عمل جمع به صورت **موازی** در یک سیکل ساعت انجام شوند، که در حجم بالای داده عملکرد را به شکل چشمگیری بهبود بخشید.

۸.۲. دستاوردها و یادگیری‌ها

مهم‌ترین نکات آموخته شده از این آزمایش عبارتند از:

- **تأثیر ساختار داده:** انتخاب ساختار داده مناسب بر عملکرد تأثیر مستقیم دارد؛ اگرچه آرایه‌های ساده برای عملیات سریع مناسبند، اما برای مدیریت پویای حافظه، بردارها بهتر هستند.
- **قدرت بهینه‌سازی کامپایلر:** کامپایلرهای مدرن با پرچم‌های صحیح (مانند `-O3`) می‌توانند عملکرد را به طور چشمگیری بهبود بخشند و استفاده از این پرچم‌ها ضروری است.

[12] Data Parallelism

- **تک دستور - چند داده و برداری سازی:** برای عملیات ساده و تکراری بر روی داده‌های متوالی، تکنیک SIMD بسیار مؤثر است و می‌تواند چندین عملیات را همزمان انجام دهد که یک شکل کلیدی از پردازش موازی است.
- **مقیاس‌پذیری:** مزایای بهینه‌سازی‌ها با افزایش اندازه داده (مقیاس‌پذیری) افزایش می‌یابد، در حالی که برای داده‌های کوچک، سربار می‌تواند از مزایای بهینه‌سازی پیشی گیرد.
- **تعادل بین خوانایی و عملکرد:** باید تعادلی منطقی بین عملکرد بهینه و حفظ خوانایی و قابلیت نگهداری کد ایجاد کرد.

۸.۳. کاربردهای عملی

تکنیک‌های بهینه‌سازی و برداری سازی یاد گرفته شده، در بسیاری از حوزه‌های محاسباتی با حجم بالای داده و عملیات تکراری، که نیاز به پردازش موازی دارند، کاربرد دارند:

- **پردازش تصویر:** عملیات فیلترها و تبدیلات بر روی پیکسل‌ها.
- **یادگیری ماشین:** محاسبات ماتریسی در شبکه‌های عصبی (مثل فرایند انتشار رو به جلو[13]).
- **شبیه‌سازی‌های علمی:** حل معادلات و شبیه‌سازی ذرات.
- **پردازش سیگنال:** عملیاتی مانند FFT و پیچش[14].

۹. منابع و مراجع

کتاب‌ها:

1. "Computer Organization and Design" - David Patterson & John Hennessy
2. "Parallel Programming in C with MPI and OpenMP" - Michael J. Quinn
3. "The Art of Multiprocessor Programming" - Maurice Herlihy & Nir Shavit
4. "Programming Massively Parallel Processors" - David Kirk & Wen-mei Hwu
5. "Optimizing Software in C++" - Agner Fog

[13] forward propagation

[14] convolution

مقالات و منابع آنلاین:

1. Intel Intrinsics Guide
2. OpenMP Specifications
3. GCC Optimization Options
4. C++ Reference - std::vector
5. Performance Analysis Guide for Intel Processors
6. Agner Fog's Optimization Manuals