УДК 004.41+004.42

К. Нестеренко, І. Стеценко

МЕТОД АВТОМАТИЗАЦІЇ РОЗРОБКИ БАГАТОПОТОЧНОЇ ПРОГРАМИ МОВОЮ C++ НА ПРИКЛАДІ КОНВЕРТАЦІЇ ЗОБРАЖЕНЬ У DDS ТЕКСТУРИ

Анотація: На сьогоднішній день використання багатопоточності є одним з основних методів оптимізації програмного забезпечення. Не в останню чергу це спричинено тим, що персональні комп’ютери, ноутбуки та навіть смартфони отримують все потужніше апаратне забезпечення, у тому числі процесори, кількість і потужність ядер яких ще кілька десятиліть назад вважалися абсолютно недосяжними. Під час розробки програмного забезпечення необхідно враховувати, які ресурси наявні у кінцевого користувача, і відповідно використовувати їх для найшвидшого отримання результату. У даному дослідженні пропонується метод автоматизації розробки багатопоточної програми мовою C++, що дає змогу організувати виконання задач пулом потоків з можливістю задати порядок виконання задач відносно одна одної, за допомогою механізму залежностей. Дослідження ефективності запропонованого методу виконано на прикладі розробки багатопоточної програми конвертації зображень у DDS текстури.

Ключові слова: програмне забезпечення, C++, стандартна бібліотека C++, багатопоточність

Вступ

З розвитком технологій постійно збільшується обчислювальна потужність персональних комп’ютерів, ноутбуків та телефонів звичайних користувачів. Не кажучи вже про сучасні можливості потужних серверів та дата центрів. Так вже давно стало нормою наявність у звичайних процесорах більш ніж десяти логічних ядер. Крім того, для вирішення задач з великою кількістю математичних обчислень почали все більше використовувати потужності графічних процесорів [1]. Проте одного лише приросту продуктивності обчислювальних ресурсів недостатньо. Раціональне використання обчислювальних ресурсів стає задачею інженерів програмного забезпечення. Якщо програмне рішення починає відставати за швидкодією від конкурентів, які

швидше адаптуються до потужностей нового апаратного забезпечення, воно може стати абсолютно неконкурентоспроможним і нанести суттєві збитки бізнесу.

Сучасні мови програмування пропонують різноманітні механізми для написання безпечного та стабільного багатопоточного коду. Проте використання багатопоточності не завжди гарантує досягнення прискорення обчислень. Використання декількох потоків, окрім переваги у швидкодії, накладає значні обмеження на те, яким чином має бути спроєктоване програмне забезпечення. Розробник має подбати як про безпечне використання даних для уникнення фатальних збоїв під час виконання програми, так і про те, щоб використані механізми синхронізації не знівелювали переваги використання багатопоточності.

Для досягнення бажаного результату необхідно спроєктувати систему, яка одночасно буде задовольняти вимоги як швидкодії, так і відмовостійкості. На розробку подібного рішення зазвичай витрачається досить багато часу і ресурсів розробника. І ще більше ресурсів вимагатиме модифікація вже існуючого програмного продукту з метою його оптимізації.

Метою даного наукового дослідження є автоматизація розробки багатопоточної програми, спрямована як на створення нового багатопоточного програмного продукту, так і на адаптацію вже існуючого. При цьому, запропоноване рішення якнайкраще проявляє свої переваги, коли розробнику потрібно виконувати задачі в певній залежності одна від одної. Також інтеграція методу дозволяє суттєво зменшити час розробки або модифікації програмного забезпечення. Основна ідея методу полягає в тому, щоб об’єднати ефективність використання обчислювальних ресурсів, яку пропонує шаблон проєктування Thread pool [2], з можливістю контролювати порядок виконання задач за допомогою join-методу.

Для експериментального дослідження ефективності запропонованого методу автоматизації на його основі розроблено програмне забезпечення для конвертації зображень у DDS текстури. Вибір саме цього програмного забезпечення обумовлено тим, що у більшості випадків використання DDS текстур для того чи іншого графічного програмного забезпечення, використовується значна кількість текстур і кожна з них може бути досить великого розміру. Щоб зменшити витрати часу спеціалістом на конвертацію

зображень у текстури основною вимогою для програмного забезпечення конвертації зображень є швидкодія.

Огляд існуючих рішень

C++ Standard Library

Як було зазначено вище, сучасні мови програмування пропонують різноманітні засоби для написання багатопоточного коду. Інженер має в своєму розпорядженні усі стандартні примітиви синхронізації такі як mutex, semaphore, condition variables та atomic типи даних [3]. Також наявні інтерфейс для роботи з потоками та алгоритми паралельних обчислень [4].

Основною перевагою використання стандартної бібліотеки є те, що вона надає широкі можливості розробити код, який найкраще підходить для вирішення задачі, поставленої перед інженером програмного забезпечення. Для цього інженеру потрібно провести детальний аналіз програмного забезпечення, виявити місця програми, виконання яких в кількох потоках дасть найбільший приріст швидкодії, та впровадити оптимізації, які будуть найбільш ефективними для розроблюваного програмного продукту.

Основний недолік бібліотеки є наслідком з її переваги. Якщо створюється унікальне рішення і фактично з нуля, то цілком ймовірно, що це буде найбільш витратне по ресурсам рішення і найдорожче з позиції бізнесу. До недоліку можна віднести також те, що можливості використання C++ Standard Library з’явились лише у стандарті C++11 [5], тому, якщо в проєкті присутні модулі, які не підтримують цей стандарт, то використання C++ Standard Library не є можливим.

Boost

Boost – це, напевно, найпопулярніша колекція бібліотек для багатопоточного програмування мовою C++. Її виникнення пов'язане з тим, що до стандарту C++ були включені не всі бібліотеки, які вважалися багатьма розробника необхідними. Зараз бібліотеки Boost позиціонуються як розширення до стандартної бібліотеки і залишаються актуальними не зважаючи на постійні оновлення стандарту C++.

Перевага Boost полягає в тому, що деякі проблеми, які виникають під час розробки багатопоточних програм, цими бібліотеками вже вирішені, на відміну від стандартної бібліотеки, й інженеру не доведеться витрачати на їх вирішення час. Для прикладу, функції стандартної бібліотеки rand або ctime, не є безпечними

для використання у багатопоточному середовищі, оскільки вони зберігають внутрішній стан що не синхронізується між потоками. Водночас, бібліотеки Boost гарантують безпечність їх використання [6]. До переваг Boost можна віднести також те, що вони підтримуються навіть досить ранніми версіями компіляторів. Популярність цієї колекції бібліотек обумовлює також високу ймовірність того, що він може бути інтегрований у проєкт, що розробляється.

Основний недолік використання бібліотеки Boost є аналогічним стандартній бібліотеці. Не зважаючи на те, що певні проблеми, які постануть перед розробником в ньому вже вирішені, ця бібліотека орієнтована на розробку програми з нуля і розробка все ще буде достатньо дорогою з позиції бізнесу. Також варто зазначити, що у проєктах використовується різна політика стосовно додавання сторонніх бібліотек, тому, не зважаючи на популярність, багато проєктів принципово відмовляються від використання Boost на користь використання виключно стандартної бібліотеки.

Concurrencpp

Concurrencpp - бібліотека для розробки багатопоточних програм мовою C++ з відкритим вихідним кодом [7], що дає змогу писати код без використання низькорівневих примітивів таких, як lock або conditional variable, замінюючи їх високорівневими абстракціями. Бібліотека має багато переваг. Фактично, вона пропонує готове рішення для написання багатопоточного коду, беручи на себе функції синхронізації даних та безпечного виконання коду у потоках. Використання цієї бібліотеки здатне суттєво підвищити швидкість розробки і відповідно зробити її більш дешевою, оскільки інженеру програмного забезпечення не потрібно буде вирішувати низькорівневі проблеми, а достатньо сфокусуватися на високорівневій архітектурі коду. Окрім того результуючий код буде простіше підтримувати і він буде містити менше дефектів.

Проте бібліотека Concurrencpp не позбавлена недоліків. Вона має досить обмежені можливості стосовно її модифікації під потреби розробника, що може призвести до потреби у написанні додаткових обгорток або проксі аби застосувати дану бібліотеку для конкретного програмного забезпечення.. Також бібліотека пропонує лише досить примітивний механізм встановлення залежностей між задачами, які виконуються в потоках, і в досить обмеженому вигляді дозволяє обмін даними між залежними задачами. До того ж бібліотека має досить значні вимоги до версії компіляторів та стандарту C++, що робить її інтеграцію неможливою для великої кількості проєктів.

Метод автоматизації розробки багатопоточної програми мовою C++

Вимоги до методу

Спираючись на огляд існуючих рішень, на їх переваги і недоліки, можна сформувати наступні функціональні та нефункціональні вимоги до методу автоматизації розробки багатопоточної програми. Основна функціональна вимога - ефективність використання наявних обчислювальних ресурсів. Тобто, має бути забезпечено ефективне використання таких ресурсів як потоки, при цьому час простою системи має бути мінімальним [8]. На додачу до цього метод повинен, за потреби, надати розробнику можливість гарантувати порядок виконання задач відносно одна одної, тобто встановити між ними залежності. Виконання цієї умови дасть змогу застосувати даний метод автоматизації для більшої кількості проєктів, для яких не достатньо рішень запропонованих класичними шаблонами проєктування.

Серед нефункціональних вимог можна виділити наступне. По-перше, реалізація методу має базуватися на стандартній бібліотеці C++. Не зважаючи на те, що для доступу до компонентів для багатопоточної розробки необхідна підтримка стандарту C++11, ця вимога виконується для набагато більшої кількості проєктів, ніж вимоги для Concurrencpp (див. 1.3) і взагалі не буде потребувати інтеграції сторонніх бібліотек. По-друге, метод має інкапсулювати всередині себе низькорівневий код, включаючи примітиви синхронізації, та надавати розробнику можливість використовувати його інтерфейс як високорівневу абстракцію [9]. При цьому має бути підримка кастомізації під потреби розробника, а також механізми, які дають змогу налаштувати залежності між задачами, що виконуються, а також обмін даними між ними.

Реалізація методу

Для початку введемо наступні абстракції. Абстрактний клас Job (рис. 1) буде замінювати собою об’єкт класу function, який зазвичай передається в якості аргументу для виконання в потоці. Це дає можливість розробнику застосовувати принципи об’єктно-орієнтованого програмування для побудови більш складної логіки виконання задач, а також реалізації механізму залежностей між екземплярами класу Job, або його нащадками. Абстрактний клас JobData агрегується всередині класу Job та зберігає стан, який може бути спільним для кількох задач, а також бути використаним для отримання результату після виконання задач.

Механізм залежностей дозволяє розробнику задати відносний порядок виконання задач. Цей механізм необхідний у випадках, коли виконання одної з задач має сенс лише після виконання певного набору інших задач. Тобто, коли говоримо, що задача Job A має залежність на задачу Job B, то мається на увазі, що Job A може бути виконана тоді і тільки тоді, коли Job B успішно завершила своє виконання. Тобто метод додає поведінку, якої можна було б досягнути, застосувавши механізм join. Проте суттєва відмінність від join полягає в тому, що дана реалізація не є блокуючою, тобто вона не блокує поточний потік в очікуванні завершення виконання іншого потоку. Більш того, якщо потік повністю завершив свою роботу, що є вимогою для спрацювання join, він знищується і не може бути використаний повторно. Запропонований метод натомість буде шукати альтернативні задачі, які можна виконувати в даний час, при цьому повторно використовуючи вже створений потік, що дозволяє більш ефективно використовувати ресурси та збільшити швидкодію програмного забезпечення.

```
class Job
{
public:
    virtual ~Job() = default;

    void Start();

    void AddJobData(const std::shared_ptr<JobData>& jobData);
    JobData* GetJobData();

    void AddDependency(const std::shared_ptr<Job>& job);

    bool IsReady();
    bool IsSuccessful();

protected:
    virtual void Execute() = 0;

private:
    bool m_bSuccessful { false };

    std::vector<std::shared_ptr<Job>> m_dependencies;
    std::shared_ptr<JobData> m_jobData;
};
```

Рисунок 1. Абстрактний клас Job

Для візуалізації механізму залежностей використаємо граф. Припустимо розробник виділив шість різних задач: Job A, Job B, Job C, Job D, Job E, Job F. Задачі мають наступні залежності: Job B та Job C залежать від Job A; Job D залежить від Job B; Job E залежить від Job C; Job F залежить від Job D та Job E (рис. 2). Дуги графу відображають порядок, в якому мають бути виконані задачі.

Не можливо гарантувати, яка з задач Job D чи Job E виконається раніше (оскільки вони виконуються паралельно), проте це і не важливо, оскільки завдання механізму залежностей – гарантувати, що Job F почне своє виконання лише після них обох.

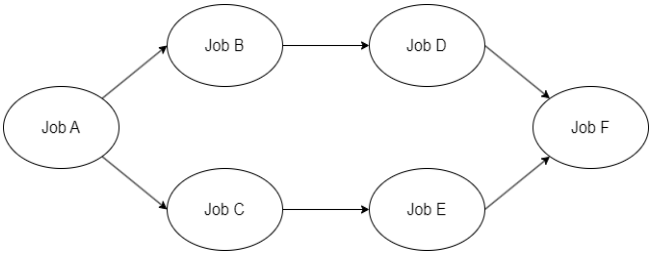

Рисунок 2. Граф залежностей між задачами

Наступним кроком реалізації методу є додавання механізму управління потоками та розподілу задач. Згідно сформованих вимог (див. 2.1) цей механізм має інкапсулювати в собі низькорівневі механізми синхронізації, та надавати розробнику високорівневий інтерфейс для взаємодії. Основою розробки є шаблон проєктування Thread Pool [2], перевагою якого є те, що потоки створюються один раз на початку створення пулу і знищуються після завершення його роботи. Це дозволяє оптимізувати використання ресурсів у випадку якщо за один запуск програмне забезпечення виконує велику кількість задач.

Проте класичний механізм упорядкування задач в черзі пулу потоків за принципом «перший увійшов, першим вийшов», має бути модифікований, щоб враховувати механізм залежностей між задачами. На рисунку 3 представлений відповідний код. Функція GetRunnableJob знаходить задачу для якої на момент перевірки вже були виконані усі її залежності і яку може виконувати вільний потік. Якщо таких задач є декілька, виконається та яка була додана у чергу раніше.

Доповнимо схему залежностей між задачами (див. рис. 2), додавши візуалізацію роботи цього механізму в певний момент часу. Припустимо метод використовує два потоки, і задачі Job A, Job C вже були виконані, а задача Job B виконується зараз в потоці Thread 1. Тоді стан задач матиме наступний вигляд (рис. 4). На схемі зеленим кольором позначені задачі які вже були виконані, а синім – ті, що виконуються в даний момент. У черзі червоним позначені задачі, залежності яких все ще не виконані і їх неможливо запустити в цей момент часу, а зеленим - відповідно ті, які готові до запуску. Як ми можемо побачити, для такого стану, не зважаючи на те, що задача Job D була додана в чергу раніше,

через те, що її залежність все ще в процесі, ми не можемо почати її виконувати, тому буде обрано задачу Job E для виконання в потоці Thread 2.

```
void JobManager::ThreadLoop()
{
    while (true)
    {
        std::shared_ptr<Job> job = nullptr;
        {
            std::unique_lock<std::mutex> lock(m_queueMutex);
            job = GetRunnableJob();

            if (!job && m_bShouldTerminate && m_jobs.size() == 0)
            {
                return;
            }
        }

        if (job)
        {
            job->Start();
        }
        else
        {
            std::this_thread::yield();
        }
    }
}

std::shared_ptr<Job> JobManager::GetRunnableJob()
{
    std::shared_ptr<Job> result = nullptr;
    for (std::vector<std::shared_ptr<Job>>::iterator it = m_jobs.begin(); it != m_jobs.end();)
    {
        if ((*it)->IsReady())
        {
            result = *it;
            m_jobs.erase(it);
            break;
        }
        else
        {
            it++;
        }
    }

    return result;
}
```

Рисунок 3. Внутрішній цикл потоку

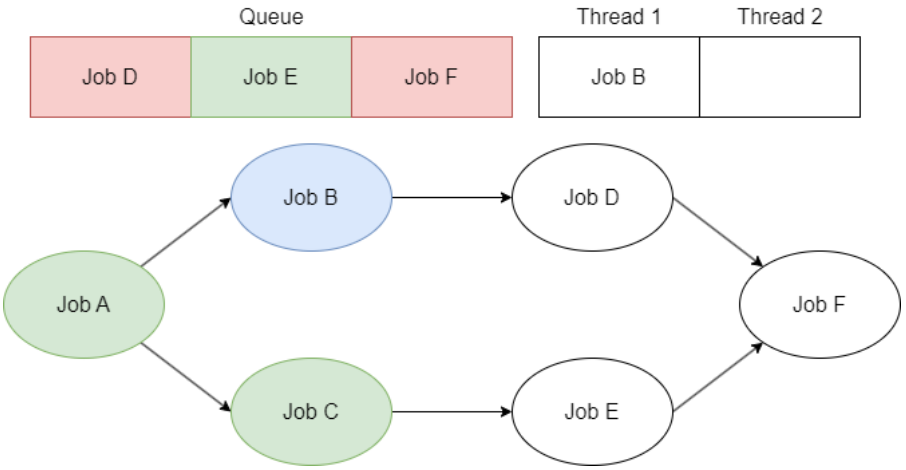

Рисунок 4. Візуальне представлення стану системи

Застосування методу

Для демонстрації роботи методу був розроблений затосунок, який конвертує звичайні зображення (png, jpg тощо) у текстури формату DDS за допомогою алгоритму BC1 [10].

Процес конвертації зображення можна розділити на два етапи. Розділення зображення на блоки розміром 4×4 пікселі, які мають назву тексель, і безпосередньо перетворення текселя у формат даних для алгоритму BC1. Відповідно ми можемо виділити два види задач і створити для них відповідні класи SplitImageJob та ConvertImageJob, які є нащадками класу Job. Очевидно, що задача ConvertImageJob має залежність від задачі SplitImageJob, тобто може бути запущена на виконання тільки після успішного завершення її виконання. Також задача SplitImageJob має передати розділене на тексели зображення та іншу інформацію про нього, яка необхідна для конвертації. Для цього створюється клас ImageConversionJobData, що є нащадком класу JobData (рис. 5).

Таким чином, замість написання низькорівневого коду, розробник використовує високорівневі абстракції та інтерфейс для виконання коду задач паралельно потоками. Це дозволяє пришвидшити розробку, а також підвищує якість вихідного коду та спрощує його довгострокову підтримку. Механізми синхронізації інкапсульовані всередині коду запропонованого метода і гарантують безпечність виконання коду. Також варто зазначити, що метод забезпечує управління ресурсами, задіяними під час його роботи. Після завершення виконання вони автоматично будуть звільнені, що унеможливлює витоки пам’яті.

```
JobManager* jobManager = new JobManager();

std::shared_ptr<JobData> jobData = std::make_shared<ImageConversionJobData>(inputFile, outputFile, DDS::CompressionType::DXT1);

std::shared_ptr<Job> splitJob = std::make_shared<SplitImageJob>(inputFile);
splitJob->AddJobData(jobData);

std::shared_ptr<Job> convertJob = std::make_shared<ConvertImageJob>();
convertJob->AddJobData(jobData);
convertJob->AddDependency(splitJob);

jobManager->QueueJob(splitJob);
jobManager->QueueJob(convertJob);

jobManager->Start();
jobManager->Stop();
```

Рисунок 5. Приклад використання методу у клієнтському коді програми

У підсумку, можемо стверджувати, що запропонований метод виконує сформульовані у підрозділі 2.1 вимоги та дає змогу значно пришвидшити розробку багатопоточного програмного забезпечення за рахунок автоматизації, відповідно зробивши його дешевшим з точки зору бізнесу.

Висновки

Розроблено метод автоматизації розробки багатопоточного програмного забезпечення мовою C++, який найкраще демонструє себе у проєктах, де необхідно мати можливість виконувати задачі в певній послідовності, при цьому оптимально використовуючи наявні ресурси програми. Для розробки методу були досліджені існуючі рішення, виділені їх основні переваги та недоліки.

Запропонований метод надає можливість розробнику за допомогою високорівневих абстракцій розбити виконання коду на окремі задачі, задати порядок виконання цих задач, та асинхронно їх виконати. При цьому для роботи методу достатньо підтримки проєктом стандарту C++11 і немає потреби в інтеграції інших сторонніх бібліотек. Для демонстрації ефективності застосування методу розроблено невеликий застосунок для конвертації зображень у DDS текстури.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. NVIDIA CUDA. NVIDIA Documentation Center | NVIDIA Developer. URL: https://docs.nvidia.com/cuda/ (дата звернення: 07.02.2023).

2. C++ Thread Pool. EDUCBA. URL: https://www.educba.com/c-plus-plus-thread-pool/ (дата звернення: 07.02.2023).

3. C++ Concurrency support library. cppreference.com. URL: https://en.cppreference.com/w/cpp/thread (date of access: 07.02.2023).

4. C++ Parallel execution. cppreference.com. URL: https://en.cppreference.com/w/cpp/algorithm/execution_policy_tag_t (дата звернення: 07.02.2023).

5. Modern multithreading and concurrency in C++. Educative. URL: https://www.educative.io/blog/modern-multithreading-and-concurrency-in-cpp (date of access: 07.02.2023).

6. Boost.Threads. Boost C++ Libraries. URL: https://www.boost.org/doc/libs/1_31_0/libs/thread/doc/overview.html (дата звернення: 07.02.2023).

7. GitHub - David-Haim/concurrencpp. GitHub. URL: https://github.com/David-Haim/concurrencpp (дата звернення: 07.02.2023).

8. An Empirical Study on C++ Concurrency Constructs / D. Wu et al. IEEE Xplore. URL: https://ieeexplore.ieee.org/document/7321187 (date of access: 07.02.2023).

9. Structured Concurrency in C++. ACCU. URL: https://accu.org/journals/overload/30/168/teodorescu/ (дата звернення: 07.02.2023).

10. Block Compression. Microsoft Learn. URL: https://learn.microsoft.com/en-us/windows/win32/direct3d10/d3d10-graphics-programming-guide-resources-block-compression#bc1 (дата звернення: 07.02.2023).