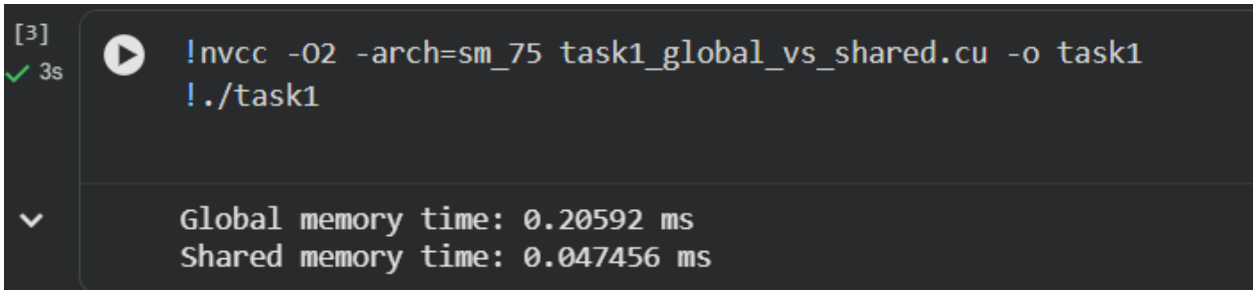
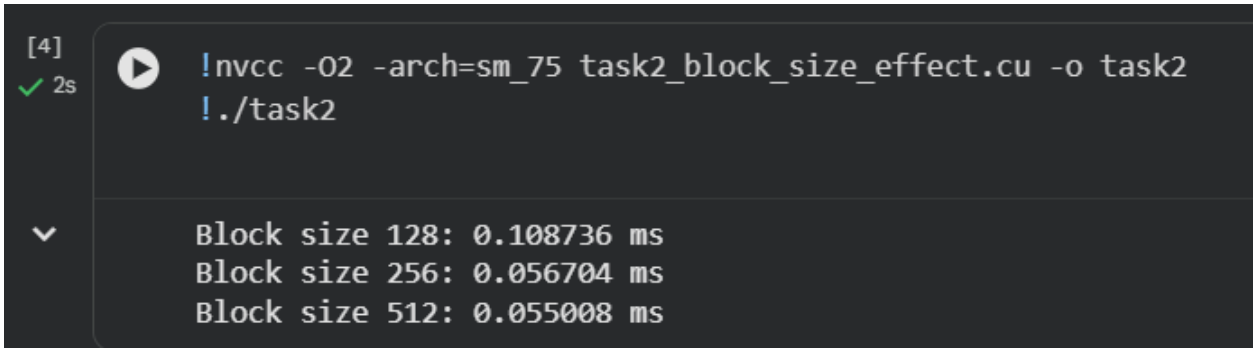
Результаты:

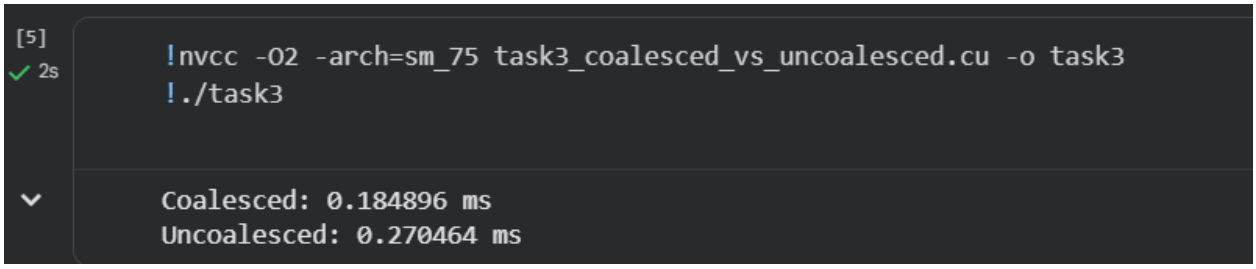
Task 1



Task 2



Task 3



Task 4

```
[6]
✓ 2s
!nvcc -O2 -arch=sm_75 task4_optimized_config.cu -o task4
!./task4

Task4: optimizing launch configuration for vector_add
N=1000000, averaging iters=10

block   grid    avg_time_ms
64      15625   0.052688
128     7813    0.0517216
256     3907    0.0518944
512     1954    0.0519392
1024    977     0.053104

Best block size: 128 (avg 0.0517216 ms)
Bad  block size: 64 (avg 0.0525536 ms)
Speedup (bad / best): 1.01609x
Correctness check: OK
```

Ответы на контрольные вопросы:

1) Какие основные типы памяти существуют в архитектуре CUDA и чем они отличаются по скорости доступа?

В CUDA обычно выделяют:

- Регистры (registers)

Самая быстрая память. Находится “при потоке”. Используется для временных переменных. Ограничена по объёму. Если регистров не хватает, часть переменных “проливается” в локальную память.

- Локальная память (local memory)

Логически “частная” для потока, но физически размещается в глобальной памяти (DRAM). Поэтому по задержкам близка к global. Возникает автоматически, когда не хватает регистров или массивы/структуры слишком большие.

- Разделяемая память (shared memory)

Очень быстрая память, общая для потоков одного блока. Используется как буфер для повторного использования данных и взаимодействия потоков. Требует синхронизации __syncthreads().

- Глобальная память (global memory)

Большая по объёму, доступна всем потокам и CPU, но самая медленная по латентности. Производительность зависит от шаблона доступа (coalescing).

- Константная память (constant memory)

Только для чтения на GPU, небольшая, кэшируется. Эффективна, когда все потоки читают одно и то же значение/похожие адреса.

- Текстурная память (texture memory)

Тоже кэшируется и оптимизирована под пространственную локальность (часто используется для 2D/3D данных).

Скорость в общем виде:

Registers > Shared > Constant/Texture cache > Global ≈ Local

(Но итоговая скорость зависит от coalescing, кэшей, occupancy и др.)

---

2) В каких случаях использование разделяемой памяти позволяет ускорить выполнение CUDA-программы?

Shared memory даёт ускорение, когда:

1. Данные повторно используются несколькими потоками блока

Например, при редукции, свёртках, stencil-операциях.

2. Нужно уменьшить число обращений к global memory

Global медленная, а shared быстрая. Если перенести промежуточные вычисления в shared, уменьшается трафик в DRAM.

3. Нужен обмен данными между потоками

Shared — стандартный способ для “внутриблочного” обмена.

4. Есть плохой шаблон чтения из global, но можно загрузить “куском” в shared коалесцированно, а дальше работать локально.

Важно: shared memory ускоряет только если:

- объём shared не слишком большой (иначе падает occupancy),
- нет сильных bank conflicts,
- действительно уменьшается число чтений/записей в global.

---

3) Как шаблон доступа к глобальной памяти влияет на производительность GPU-программы?

Глобальная память обслуживается транзакциями (пакетами). Для максимальной пропускной способности нужно, чтобы потоки варпа (32 потока) обращались к последовательным адресам.

- Коалесцированный доступ:

потоки варпа читают/пишут соседние элементы (a[i], a[i+1], …).

Тогда контроллер памяти может обслужить это минимальным количеством транзакций → высокая bandwidth.

- Некоалесцированный доступ:

потоки читают “вразнобой” (a[i*32], произвольный stride, случайные индексы).

Тогда нужно намного больше транзакций → резко падает пропускная способность и растёт время.

Итог: один и тот же объём данных может читаться одинаково, но из-за шаблона доступа GPU тратит на это сильно разное время.

---

4) Почему одинаковый алгоритм на GPU может показывать разное время выполнения при разных способах обращения к памяти?

Потому что на GPU часто “бутылочное горлышко” — не арифметика, а память:

- различное количество транзакций global memory (coalesced vs uncoalesced),
- попадания/промахи в кэшах,
- bank conflicts в shared memory,
- разное давление на регистры (регистры → local spill → медленно),
- различная occupancy (сколько варпов активно и могут скрывать латентность памяти),
- разные конфликты ресурсов (например, слишком много shared на блок → меньше блоков на SM).

То есть алгоритм “логически одинаковый”, но фактическая микроархитектурная стоимость разная.

---

5) Как размер блока потоков влияет на производительность CUDA-ядра?

Размер блока влияет на:

1. Occupancy — сколько активных варпов может находиться на SM одновременно.

Слишком маленький блок → мало варпов → хуже скрывается латентность памяти.

Слишком большой блок → может “упереться” в регистры/shared и тоже снизить occupancy.

2. Эффективность варпов

Варп = 32 потока. Если блок не кратен 32, часть варпа простаивает.

Поэтому block sizes обычно выбирают кратные 32.

3. Использование ресурсов (регистры/shared memory)

Чем больше потоков на блок, тем больше суммарные требования.

Это может ограничить число блоков на SM.

4. Память

Иногда block size влияет на coalescing и шаблон доступа.

На практике часто тестируют: 128 / 256 / 512 (и реже 1024).

---

6) Что такое варп и почему важно учитывать его при разработке CUDA-программ?

Варп (warp) — базовая единица исполнения на NVIDIA GPU: 32 потока, которые выполняют инструкции синхронно (SIMT).

Почему важно:

- Divergence: если в варпе потоки идут по разным веткам if/else, GPU вынужден выполнять ветки по очереди → падает производительность.
- Coalescing рассчитывается на уровне варпа: важно, как именно 32 потока читают память.
- Размер блока кратный 32 обычно эффективнее.

---

7) Какие факторы необходимо учитывать при выборе конфигурации сетки и блоков потоков?

Главные факторы:

1. Кратность 32 (warp size)
2. Occupancy: сколько активных варпов/блоков на SM возможно
3. Ограничения по ресурсам:
   - регистры на поток
   - shared memory на блок
   - максимум потоков на SM/на блок

4. Шаблон доступа к памяти

5. Размер данных и работа с “хвостом”

6. Профилирование: иногда лучший block size определяется экспериментом и Nsight Compute

---

8) Почему оптимизация CUDA-программы часто начинается с анализа работы с памятью, а не с изменения алгоритма?

Потому что во многих задачах GPU простаивает, ожидая данные из памяти:

- скорость вычислений очень высокая,
- но если данные приходят медленно (global memory, плохой доступ), ядро становится memory-bound.

Поэтому сначала оптимизируют:

- coalescing,
- использование shared memory,
- уменьшение трафика в global,
- устранение лишних чтений/записей,
- уменьшение local spills (регистры),
- правильный block size и occupancy.

И только если память уже “выжата”, переходят к изменению алгоритма или математическим оптимизациям.