

به نام خدا



دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

گزارشکار پروژه نهایی برنامه‌نویسی چندکرnel‌ای

زهرا لطیفی 9923069

آیدا احمدی پارسا 9923003

توضیح کد مربوط به کرنل‌های نوشته شده

```
// Define the CUDA kernel for convolution
__global__ void conv2DKernel(
    const float* input, const float* filter,
    float* output, int w, int h, int c, int k,
    int filter_w, int filter_h, int pad_w, int pad_h,
    int stride_w, int stride_h, int out_w, int out_h) {

    int out_x = blockIdx.x * blockDim.x + threadIdx.x;
    int out_y = blockIdx.y * blockDim.y + threadIdx.y;
    int out_z = blockIdx.z * blockDim.z + threadIdx.z;

    if (out_x < out_w && out_y < out_h && out_z < k) {
        float sum = 0.0f;
        for (int i = 0; i < c; ++i) {
            for (int j = 0; j < filter_h; ++j) {
                for (int l = 0; l < filter_w; ++l) {
                    int in_x = out_x * stride_w - pad_w + l;
                    int in_y = out_y * stride_h - pad_h + j;
                    if (in_x >= 0 && in_x < w && in_y >= 0 && in_y < h) {
                        sum += input[(i * h + in_y) * w + in_x] * filter[((out_z
* c + i) * filter_h + j) * filter_w + l];
                    }
                }
            }
        }
        output[(out_z * out_h + out_y) * out_w + out_x] = sum;
    }
}

void cuConv2D(float *input, float *output, int w, int h, int c, int n, int k,
             int filter_w, int filter_h, int dilation_w, int dilation_h,
             int pad_w, int pad_h, int wstride, int hstride) {

    int out_w = (w + 2 * pad_w - dilation_w * (filter_w - 1) - 1) / wstride + 1;
    int out_h = (h + 2 * pad_h - dilation_h * (filter_h - 1) - 1) / hstride + 1;

    float *h_filter = (float *)malloc(k * c * filter_w * filter_h *
sizeof(float));
    for (int i = 0; i < k * c * filter_w * filter_h; i++) {
        h_filter[i] = (float)std::rand() / RAND_MAX / 1000;
    }

    // Allocate GPU memory
```

```

float *d_input, *d_filter, *d_output;
cudaMalloc(&d_input, n * c * h * w * sizeof(float));
cudaMalloc(&d_filter, k * c * filter_w * filter_h * sizeof(float));
cudaMalloc(&d_output, n * k * out_h * out_w * sizeof(float));

    cudaMemcpy(d_input, input, n * c * h * w * sizeof(float),
cudaMemcpyHostToDevice);
    cudaMemcpy(d_filter, h_filter, k * c * filter_w * filter_h * sizeof(float),
cudaMemcpyHostToDevice);

// Define block and grid sizes
int blockSize = 16;
dim3 threadsPerBlock(blockSize, blockSize, 1);
dim3 numBlocks((out_w + blockSize - 1) / blockSize, (out_h + blockSize - 1) /
blockSize, k);

auto start = std::chrono::steady_clock::now();

// Launch the custom convolution kernel
conv2DKernel<<<numBlocks, threadsPerBlock>>>(d_input, d_filter, d_output, w,
h, c, k, filter_w, filter_h, pad_w, pad_h, wstride, hstride, out_w, out_h);
cudaDeviceSynchronize();

auto end = std::chrono::steady_clock::now();

int fwd_time = static_cast<int>(std::chrono::duration<double, std::micro>(end -
start).count());
std::cout << " " << fwd_time << " ms" << std::endl;

cudaMemcpy(output, d_output, n * k * out_h * out_w * sizeof(float),
cudaMemcpyDeviceToHost);

// Free device memory
cudaFree(d_input);
cudaFree(d_filter);
cudaFree(d_output);
free(h_filter);
}

```

اولین بخش جایگزین شده، تابع مربوط به کانولوشن دو بعدی است که با یک کرنل و یک تابع برای انتقال بین GPU و CPU به روز شده.

تابع کرنل conv2DKernel

این تابع بر روی GPU اجرا می شود و عملیات کانولوشن را انجام می دهد.

- indexing - نخ: هر نخ شاخص خروجی خود (`out_z, out_y, out_x`) را بر اساس موقعیتش در گردید و بلوک محاسبه می کند.

- بررسی مرز: اطمینان حاصل می کند که نخ با یک موقعیت خروجی معتبر مطابقت دارد.

- کانولوشن: روی کانال های ورودی (`C`), و عرض فیلتر (`filter_w`) و ارتفاع (`filter_h`) تکرار می شود. برای هر موقعیت، شاخص های ورودی مربوطه (`in_y, in_x`) را با در نظر گرفتن گام و `padding` محاسبه می کند. اگر این اندیس ها در محدوده تصویر ورودی باشند، عملیات ضرب و مجموع را انجام می دهد.

- نتیجه کانولوشن در آرایه خروجی در شاخص خروجی محاسبه شده نوشته می شود.

تابع `:host cuConv2D` سمت

محاسبه اندازه خروجی: عرض خروجی (`out_w`) و ارتفاع (`out_h`) را بر اساس ابعاد ورودی، `padding`، گام، محاسبه می کند.

یک فیلتر را با مقادیر تصادفی تعریف می کند و مقداردهی اولیه می کند.

تخصیص حافظه GPU: حافظه را روی GPU برای ورودی، فیلتر و خروجی تخصیص می دهد.

انتقال داده: داده های ورودی و فیلتر را از `device host` (CPU) به `(GPU)` کپی می کند.

اندازه بلوک های نخ و گردید را مشخص می کند.

«conv2DKernel» را فراخوانی می کند.

همگام سازی: منتظر می ماند تا پردازش گرافیکی برای تمام نخ ها به پایان برسد.

زمان صرف شده برای عملیات کانولوشن را هم اندازه گیری و مشارکت می کند.

نهایتاً داده های خروجی را از دستگاه به هاست کپی می کند.

Memory Cleanup GPU و حافظه میزبان اختصاص داده شده را آزاد می کند.

عملیات کانولوشن سنگ بنای بسیاری از الگوریتم های یادگیری عمیق است، به ویژه در شبکه های عصبی کانولوشن (CNN)، جایی که برای پردازش داده های تصویر استفاده می شود. استفاده از بلوک ها و نخ ها باعث می شود که کانولوشن به صورت موازی برای بخش های مختلف خروجی محاسبه شود، که به طور قابل توجهی سرعت فرآیند را در مقایسه با محاسبات متواالی در یک CPU افزایش می دهد.

برای پیاده سازی بخش مربوط به ضرب ماتریس، هر سه انواع پیاده سازی ضرب ماتریسی در تمرین سری ششم بررسی شد اما نهایتاً ضرب عادی خروجی بهتری داشت:

```
__global__ void matrixMulKernel(const float* A, const float* B, float* C, int m,
int n, int k) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m && col < n) {
        float value = 0.0f;
        for (int e = 0; e < k; ++e) {
            value += A[row * k + e] * B[e * n + col];
        }
        C[row * n + col] = value;
    }
}

void cuFC(float *input, float *output, int left, int right) {
    int m = 1, k = left, n = right;

    float *h_B = (float *)malloc(left * right * sizeof(float));
    for (int i = 0; i < left * right; i++) {
        h_B[i] = (float)std::rand() / RAND_MAX / 1000;
    }

    // Allocate 3 arrays on GPU
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, left * sizeof(float));
    cudaMalloc(&d_B, left * right * sizeof(float));
    cudaMalloc(&d_C, right * sizeof(float));

    cudaMemcpy(d_A, input, left * sizeof(float), cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, left * right * sizeof(float), cudaMemcpyHostToDevice);

    // Define block and grid sizes
    int blockSize = 16; // Adjust as necessary
    dim3 threadsPerBlock(blockSize, blockSize);
    dim3 numBlocks((n + blockSize - 1) / blockSize, (m + blockSize - 1) /
blockSize);

    auto start = std::chrono::steady_clock::now();

    // Launch the custom matrix multiplication kernel
    matrixMulKernel<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C, m, n, k);
    cudaDeviceSynchronize();
```

```

    auto end = std::chrono::steady_clock::now();

    int fwd_time = static_cast<int>(std::chrono::duration<double, std::micro>(end - start).count());
    std::cout << " " << fwd_time << " ms" << std::endl;

    cudaMemcpy(output, d_C, right * sizeof(float), cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    free(h_B);
}

```

تابع کرنل matrixMulKernel :

این تابع بر روی GPU اجرا می شود و عملیات ضرب ماتریس را انجام می دهد.

ایندکسینگ نخ: هر نخ شاخص های خود («ردیف» و «col») را بر اساس موقعیتش در شبکه و بلوک محاسبه می کند.

بررسی مرز: اطمینان حاصل می کند که نخ با یک موقعیت معتبر در ابعاد ماتریس خروجی مطابقت دارد.

ضرب ماتریس: یک "value" محلی را برای جمع آوری نتیجه تعریف می کند. سپس روی بعد مشترک ("k") تکرار می شود و حاصل ضرب نقطه ای سطر "A" و ستون "B" را انجام می دهد و نتیجه را در "value" جمع می کند.

مقدار محاسبه شده در ماتریس خروجی "C" در موقعیت مربوط به "ردیف" و "col" فعلی نوشته می شود.

تابع میزبان: «cuFC

این تابع بر روی CPU اجرا می شود و عملیات ضرب ماتریس را تنظیم می کند. در اینجا.

ابعاد ماتریس: ابعاد ماتریس های دخیل در ضرب را تعریف می کند ($n \times k \times m$) که در آن m روی ۱ تنظیم می شود که به معنای یک ماتریس یا بردار تک ردیفی است.

تخصیص حافظه میزبان: حافظه را بر روی میزبان برای ماتریس "B" تخصیص می دهد و آن را با مقادیر تصادفی مقداردهی اولیه می کند.

تخصیص حافظه GPU: حافظه را بر روی GPU برای ماتریس های A، B و C اختصاص می دهد.

انتقال داده: داده های ورودی ماتریس های A و B را از میزبان به device کپی می کند.

راه اندازی کرنل: «matrixMulKernel» را با شبکه و اندازه بلوک فراخوانی می‌کند و منتظر می‌ماند تا پردازش گرافیکی به پایان برسد.

در آخر زمان صرف شده برای عملیات ضرب ماتریس را اندازه گیری می‌کند و ماتریس خروجی C را از دستگاه به میزبان کپی می‌کند.

GPU Memory Cleanup: حافظه میزبان اختصاص داده شده را آزاد می‌کند.

:MAXPool

```
__global__ void maxPoolKernel(const float *input, float *output, int w, int h,
int c, int n, int pool_w, int pool_h, int stride_w, int stride_h) {
    int out_w = w / stride_w;
    int out_h = h / stride_h;

    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int total_elems = n * c * out_h * out_w;

    if (idx < total_elems) {
        int out_x = idx % out_w;
        int out_y = (idx / out_w) % out_h;
        int channel = (idx / (out_w * out_h)) % c;
        int batch = idx / (out_w * out_h * c);

        int start_x = out_x * stride_w;
        int start_y = out_y * stride_h;

        // Shared memory for pooling
        extern __shared__ float shared_input[];
        float* shared_channel = &shared_input[threadIdx.z * pool_h * pool_w];

        // Load input data into shared memory
        for (int i = 0; i < pool_h; ++i) {
            for (int j = 0; j < pool_w; ++j) {
                int cur_x = start_x + j;
                int cur_y = start_y + i;
                int input_idx = ((batch * c + channel) * h + cur_y) * w + cur_x;
                shared_channel[i * pool_w + j] = (cur_x < w && cur_y < h) ?
input[input_idx] : -FLT_MAX;
            }
        }
    }
}
```

```

    __syncthreads(); // Ensure all threads have loaded data into shared
memory

    // Compute max value within the pooling window
    float max_val = -FLT_MAX;
    for (int i = 0; i < pool_h; ++i) {
        for (int j = 0; j < pool_w; ++j) {
            max_val = fmaxf(max_val, shared_channel[i * pool_w + j]);
        }
    }

    // Store result in output
    int output_idx = ((batch * c + channel) * out_h + out_y) * out_w + out_x;
    output[output_idx] = max_val;
}

void cuMaxPool(float *input, float *output, int w, int h, int c, int n) {
    const int pool_w = 2;
    const int pool_h = 2;
    const int stride_w = 2;
    const int stride_h = 2;

    int out_w = w / stride_w;
    int out_h = h / stride_h;

    float *d_input, *d_output;

    // Allocate memory on the device
    cudaMalloc(&d_input, n * c * h * w * sizeof(float));
    cudaMalloc(&d_output, n * c * out_h * out_w * sizeof(float));

    // Copy input data from host to device
    cudaMemcpy(d_input, input, n * c * h * w * sizeof(float),
cudaMemcpyHostToDevice);

    // Launch the custom max pooling kernel
    int total_elems = n * c * out_h * out_w;
    int blockSize = 256;
    int numBlocks = (total_elems + blockSize - 1) / blockSize;

    auto start = std::chrono::steady_clock::now();

```

```

    maxPoolKernel<<<numBlocks, blockSize>>>(d_input, d_output, w, h, c, n,
pool_w, pool_h, stride_w, stride_h);
    cudaDeviceSynchronize();
    auto end = std::chrono::steady_clock::now();

    // Copy output data from device to host
    cudaMemcpy(output, d_output, n * c * out_h * out_w * sizeof(float),
cudaMemcpyDeviceToHost);

    // Measure and print the elapsed time
    int fwd_time = static_cast<int>(std::chrono::duration<double, std::micro>(end -
start).count());
    std::cout << " " << fwd_time << " ms" << std::endl;

    // Free device memory
    cudaFree(d_input);
    cudaFree(d_output);
}

```

:تابع کرنل CUDA است که عملیات maxpooling را انجام می‌دهد.

- عرض خروجی (out_w) و ارتفاع (out_h) را بر اساس ابعاد ورودی و گام محاسبه می‌کند.
- هر نخ شاخص منحصر به فرد خود (idx^i) را محاسبه می‌کند و روی عنصر خاصی از آرایه خروجی کار می‌کند.
- موقعیت (out_y, out_x) را در آرایه خروجی، کانال (channel) و دسته (batch) که مسئول آن است را تعیین می‌کند.
- کرنل پنجره‌ای از داده‌های ورودی را با در نظر گرفتن اندازه گام ("pool_w", "pool_h", "stride_w", "stride_h") و بارگذاری می‌کند.
- برای اطمینان از بارگیری داده‌ها، همه نخ‌ها را همگام می‌کند ($\text{syncthreads_}()$).
- هر نخ حداکثر مقدار را در پنجره خود محاسبه می‌کند و نتیجه را در آرایه خروجی می‌نویسد.

:تابع سمت host است که عملیات را تنظیم می‌کند.

- اندازه و گام پنجره را مشخص می‌کند.
- حافظه را روی GPU برای داده‌های ورودی و خروجی (cudaMalloc) اختصاص می‌دهد.
- داده‌های ورودی را از میزبان به GPU کپی می‌کند ("cudaMemcpy").
- تعداد بلوك‌ها و نخ‌ها را برای راهاندازی کرنل با آن محاسبه می‌کند ("blockSize", "numBlocks").

- زمان شروع را ثبت می‌کند، کرنل را راه اندازی می‌کند و منتظر می‌ماند تا کار GPU تمام شود.
(.«cudaDeviceSynchronize»)
- زمان پایان را ثبت می‌کند و زمان سپری شده برای عملیات را محاسبه می‌کند.
- داده‌های خروجی حاصل را به host کپی می‌کند.
- حافظه GPU اختصاص داده شده ("cudaFree") را آزاد می‌کند.

عملیات MAXPooling، ابعاد (عرض و ارتفاع) داده‌های ورودی را با اعمال یک فیلتر حداکثر در مناطق فرعی غیر همپوشان تعريف شده توسط اندازه و گام کاهش می‌دهد. این معمولاً برای نمونه‌برداری از نقشه‌های ویژگی در شبکه‌های عصبی، کاهش میزان محاسبات و کمک به تشخیص ویژگی‌ها در تغییر مقیاس و جهت‌گیری استفاده می‌شود. این کد همچنانی شامل قابلیت زمان بندی برای اندازه گیری عملکرد عملکرد GPU است.

در این کد استفاده از حافظه مشترک به کاهش تأخیر حافظه و افزایش توان عملیاتی کمک می‌کند. اعلان "خارجی __shared" برای اعلام حافظه مشترک با اندازه پویا استفاده می‌شود که سریعتر از حافظه global است و در بین نخهای موجود در همان بلوک به اشتراک گذاشته می‌شود.تابع fmaxf برای محاسبه حداکثر دو عدد ممیز شناور که برای عملیات ضروری است استفاده می‌شود. موارد توضیح داده شده تمام تغییراتی بود که ما در کد اعمال کردیم. حال یک بار کد اصلی و دیگریار کد خود را اجرا کرده و زمان‌ها را گزارش کردیم:

CONV 224x224x64 2227 ms	CONV 224x224x64 3895 ms
CONV 224x224x64 277 ms	CONV 224x224x64 79799 ms
POOLMAX 112x112x64 19477 ms	POOLMAX 112x112x64 585 ms
CONV 112x112x128 776 ms	CONV 112x112x128 20668 ms
CONV 112x112x128 1060 ms	CONV 112x112x128 39019 ms
POOLMAX 56x56x128 888 ms	POOLMAX 56x56x128 339 ms
CONV 56x56x256 1431 ms	CONV 56x56x256 13304 ms
CONV 56x56x256 1551 ms	CONV 56x56x256 23671 ms
POOLMAX 28x28x256 1345 ms	POOLMAX 28x28x256 226 ms
CONV 28x28x512 1072 ms	CONV 28x28x512 7915 ms
CONV 28x28x512 1105 ms	CONV 28x28x512 14923 ms
POOLMAX 14x14x512 920 ms	POOLMAX 14x14x512 378 ms
CONV 14x14x1024 398 ms	CONV 14x14x1024 6706 ms
CONV 14x14x1024 634 ms	CONV 14x14x1024 12402 ms
POOLMAX 7x7x1024 651 ms	POOLMAX 7x7x1024 291 ms
FC 4096 25760 ms	FC 4096 83719 ms
FC 4096 3548 ms	FC 4096 6836 ms
FC 1000 1146 ms	FC 1000 3593 ms

اعداد سمت چپ مربوط به کد اصلی و سمت راست مربوط به کد ماست که شاهدیم تا حد خوبی سعی شده سرعت حفظ شود.

مربوط به ضرب ماتریس ۳۲ در ۳۲ در ۲۰۰ در ۲۱ Performance PoolMax تا ۲۰۰ درصد و مجموع زمان اجرا برابر ۲۱ درصد بود.

پروفایل و تجزیه و تحلیل

در این بخش، به پروفایل کردن و تجزیه و تحلیل دقیق کرنل‌های CUDA با استفاده از NVIDIA Nsight Compute و Nsight Systems می‌پردازیم. پروفایل برای درک ویژگی‌های عملکرد پیاده‌سازی CUDA ما، شناسایی گلوگاه‌ها و بهینه‌سازی‌ها بسیار مهم است.

Nsight Systems و NVIDIA Nsight Compute پروفایل کردن با

پروفایل کردن به ما کمک می‌کند تا مشکلات عملکرد را مشخص کنیم و کرنل‌های CUDA را برای کارایی بهتر بهینه کنیم. Nsight Systems و NVIDIA Nsight Compute ابزارهای قدرتمندی هستند که این فرآیند را تسهیل می‌کنند. برای GPU‌هایی که از پروفایلرهای جدید پشتیبانی نمی‌کنند، NVIDIA Visual Profiler می‌تواند به عنوان جایگزین استفاده شود.

پروفایل کردن مراحل و نتایج

مدت زمان کرنل و توان عملیاتی

نمایش اجمالی پروفایل‌سازی کرنل: برای شروع، ما از NVIDIA Nsight Compute برای پروفایل کردن کرنل‌های CUDA خود استفاده می‌کنیم. این شامل اجرای برنامه CUDA ما در پروفایل برای جمع‌آوری داده‌های عملکرد است.

تحلیل خلاصه صفحه:

مدت کرنل: در صفحه خلاصه کل زمان اجرای هر کرنل را ثبت می‌کنیم. این به ما کمک می‌کند تا کرنل‌هایی را که دارای زمان برتر و ارضا کننده اهداف اصلی بهینه‌سازی هستند، شناسایی کنیم.

Compute throughput محاسباتی GPU: ما توان محاسباتی را برای هر کرنل اندازه‌گیری می‌کنیم، که نشان می‌دهد کرنل چقدر از منابع GPU به طور مؤثر استفاده می‌کند.

Throughput حافظه: ما همچنین میزان توان عملیاتی حافظه را یادداشت می‌کنیم، که نشان می‌دهد کرنل چقدر از پهنای باند حافظه GPU به طور مؤثر استفاده می‌کند.

طولانی‌ترین زمان اجرا: با شناسایی کرنل‌ای با طولانی‌ترین زمان اجرا، آن را برای تجزیه و تحلیل دقیق و بهینه‌سازی در اولویت قرار می‌دهیم.

GPU Speed Of Light Throughput Tab

محاسبات و توان عملیاتی حافظه:

- در برگه GPU Speed Of Light Throughput، هم میزان محاسبه و هم توان حافظه را برای هر کرنل بررسی می‌کنیم.

• Compute throughput با مقایسه توان محاسباتی با حداکثر نظری، تعیین می‌کنیم که آیا کرنل به طور کامل از منابع محاسباتی GPU استفاده می‌کند یا خیر.

• Throughput حافظه: به طور مشابه، ما ظرفیت حافظه را با حداکثر پهنای باند تئوری حافظه مقایسه می‌کنیم تا کارایی را ارزیابی کنیم.

• تحلیل کران: این مرحله به ما کمک می‌کند تعیین کنیم که کرنل محاسباتی (محدود شده توسط قدرت پردازش) است یا محدود به حافظه (محدود به سرعت دسترسی به حافظه).

: roofline نمودار

• نمودار روفلاین به صورت بصری تعادل بین محاسبات و توان حافظه را نشان می‌دهد.

• موقعیت نمودارها: ما کرنل خود را روی نمودار روفلاین قرار می‌دهیم تا کارایی آن را ارزیابی کنیم.

• تحلیل بهبود: اگر کرنل از روفلاین دور باشد نشان‌دهنده ناکارآمدی است. سپس تحلیل می‌کنیم که آیا بهینه‌سازی عملیات محاسباتی یا دسترسی‌های حافظه می‌تواند کرنل را به روفلاین نزدیک‌تر کند یا خیر.

• محاسبه در مقابل کران حافظه: نمودار روفلاین کمک می‌کند تا تجزیه و تحلیل ما را در مورد اینکه کرنل محاسباتی یا محدود به حافظه است تأیید کنیم.

برگه تحلیل حجم کاری حافظه

تحلیل میزان مصرف حافظه:

• در تب Memory Workload Analysis به بررسی توزیع استفاده از حافظه مشترک و جهانی می‌پردازیم.

• نمودار استفاده از حافظه: نمودار میزان مصرف حافظه، نسبت حافظه مشترک به حافظه جهانی را نشان می‌دهد. استفاده زیاد از حافظه جهانی نسبت به حافظه مشترک ممکن است نشان‌دهنده فرسته‌های بهینه‌سازی باشد.

• Cache Hit-Rates: ما نرخ ضربه کش را برای سطوح مختلف کش (L1, L2 و غیره) بررسی می‌کنیم. نرخ ضربه کم حافظه پنهان می‌تواند منجر به تاخیر بیشتر و کاهش عملکرد شود.

هشدارها و توصیه‌ها:

• ما به دنبال اخطارهای مربوط به دسترسی‌های حافظه بدون ادغام هستیم که می‌تواند به طور قابل توجهی توان حافظه را کاهش دهد.

• پروفایلر اغلب توصیه‌هایی را برای رسیدگی به مسائل عملکرد خاص ارائه می‌دهد و ما را به سمت بهینه‌سازی‌های مؤثر راهنمایی می‌کند.

تب Occupancy

تحليل Occupancy

- در تب Occupancy، نرخ Occupancy نظری و دستیافته را با هم مقایسه می‌کنیم.
- Occupancy نظری: این نشان‌دهنده حداکثر تعداد ممکن تابهای فعال در یک چند پردازنه است.
- به‌دست‌آمده: این تعداد واقعی تابهای فعال در طول اجرای کرنل است.
- تحلیل عدم تطابق: با شناسایی اختلافات بین Occupancy نظری و Occupancy شده، می‌توانیم مواردی مانند استفاده بیش از حد از رجیستر یا موازی‌سازی ناکافی را مشخص کنیم.

مراحل بهینه‌سازی:

- برای بهبود Occupancy، کاهش استفاده از ثبت، بهینه‌سازی اندازه بلوك نخ و افزایش موازی بودن را در نظر می‌گیریم.

خلاصه تجزیه و تحلیل

محاسبات در مقابل محدودیت حافظه:

- ما یافته‌های خود را در مورد اینکه آیا هر کرنل محدود به محاسبات است یا محدود به حافظه بر اساس تحلیل توان و روفلاین خلاصه می‌کنیم.

فرصت‌های بهینه‌سازی:

- بر اساس داده‌های پروفایل، ما مناطق خاصی را برای بهبود توان محاسباتی و حافظه شناسایی می‌کنیم. این ممکن است شامل بهینه‌سازی الگوهای دسترسی به حافظه، افزایش استفاده از حافظه پنهان، کاهش استفاده از ثبت یا افزایش موازی‌سازی باشد.

بهبود عملکرد:

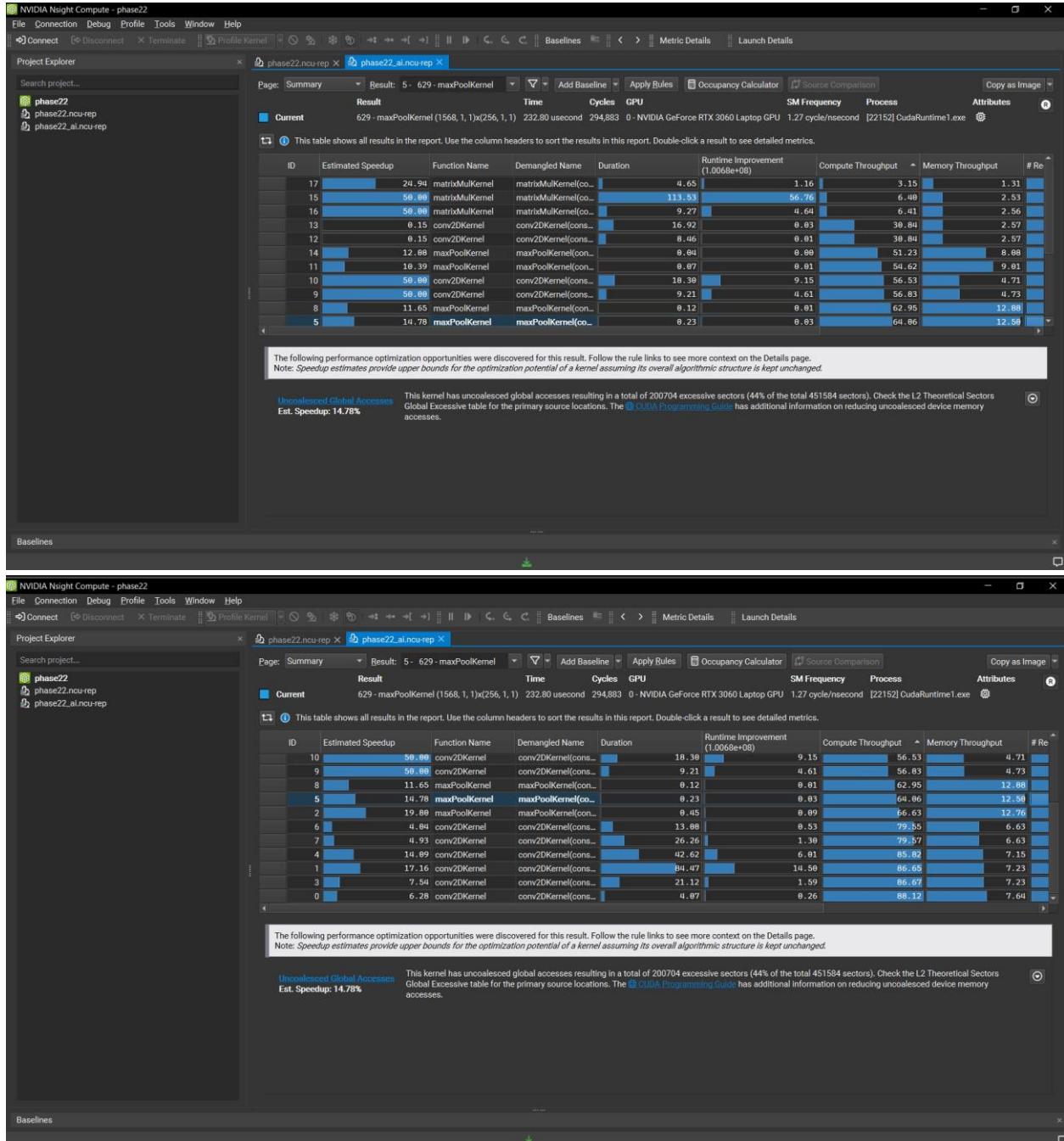
- ما در مورد بهبود عملکرد حاصل از این بهینه‌سازی‌ها با استفاده از معیارهای کمی برای نشان دادن تأثیر بحث می‌کنیم.

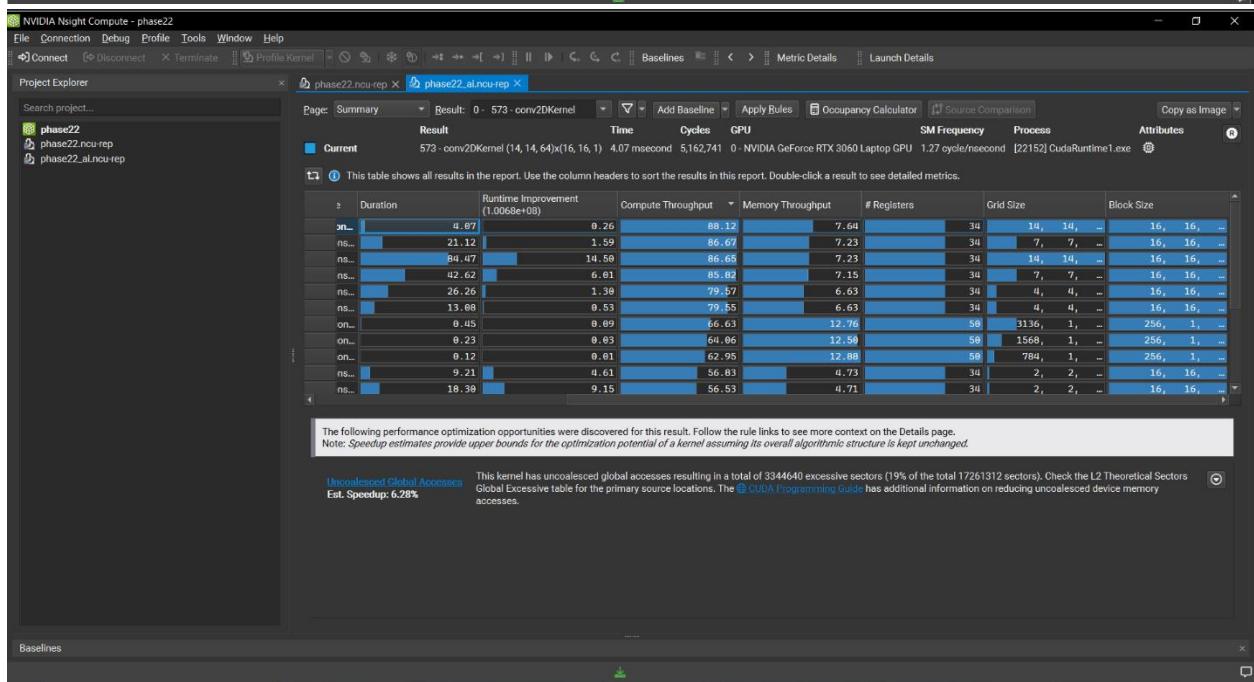
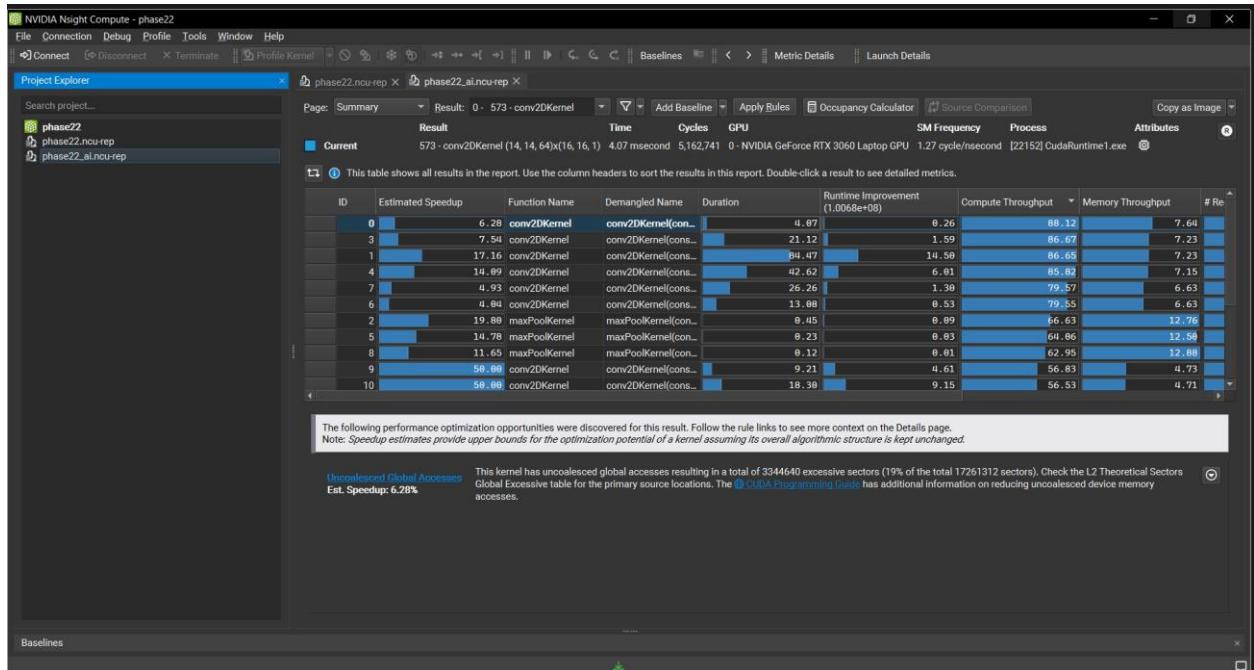
خلاصه ابزارهای پروفایل

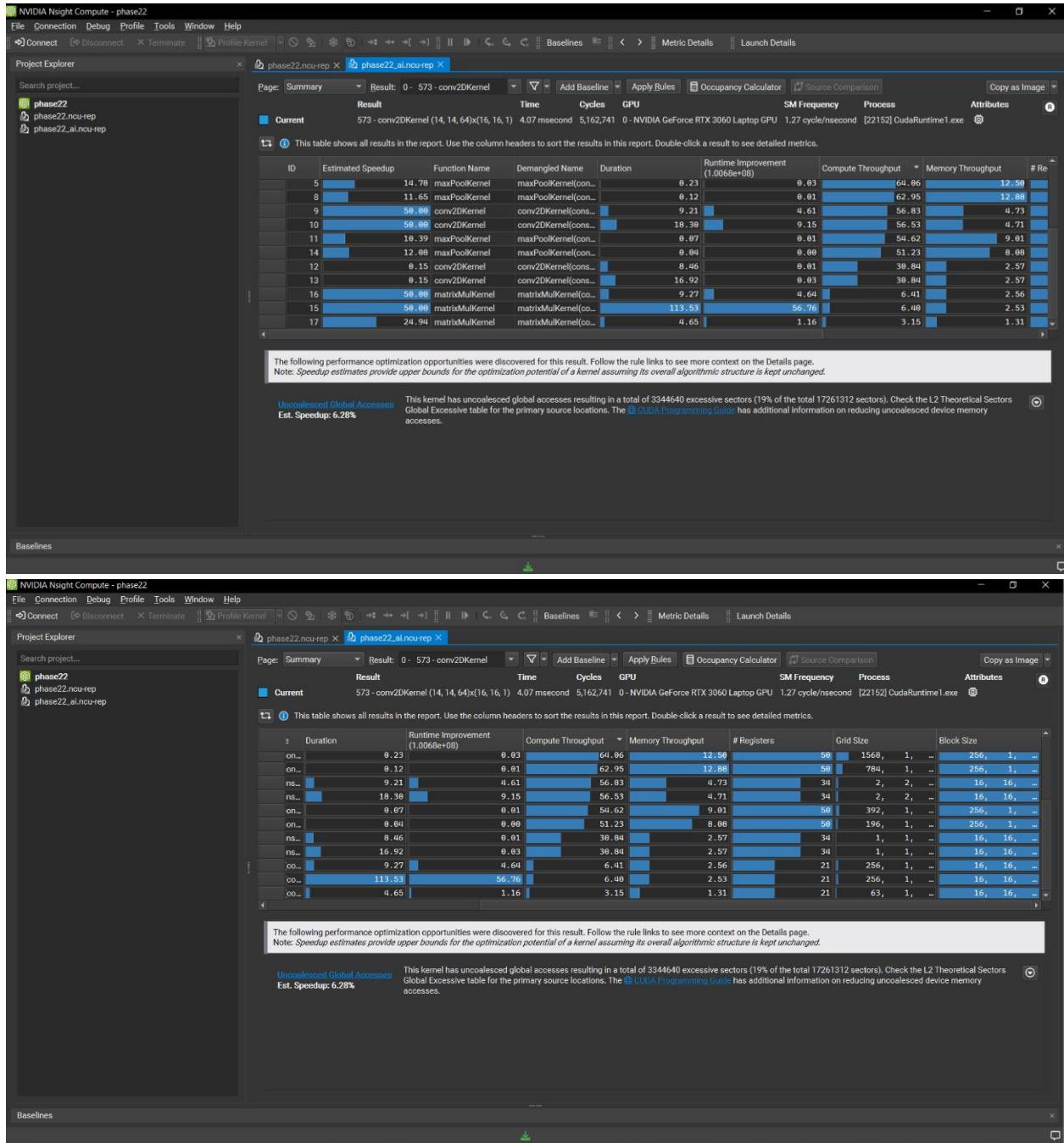
NVIDIA Nsight Compute: این ابزار تجزیه و تحلیل دقیق عملکرد کرنل، الگوهای دسترسی به حافظه و میزان مشغولیت را رائه می‌دهد.

NVIDIA Nsight Systems: این ابزار نمای وسیع‌تری از عملکرد برنامه ارائه می‌دهد و نشان می‌دهد که کرنل‌های مختلف چگونه با هم تعامل دارند و زمان صرف شده در کل برنامه کجاست.

- In the summary page, take note of all kernel call's duration, compute throughput and memory throughput. Which kernel took the longest time to execute?

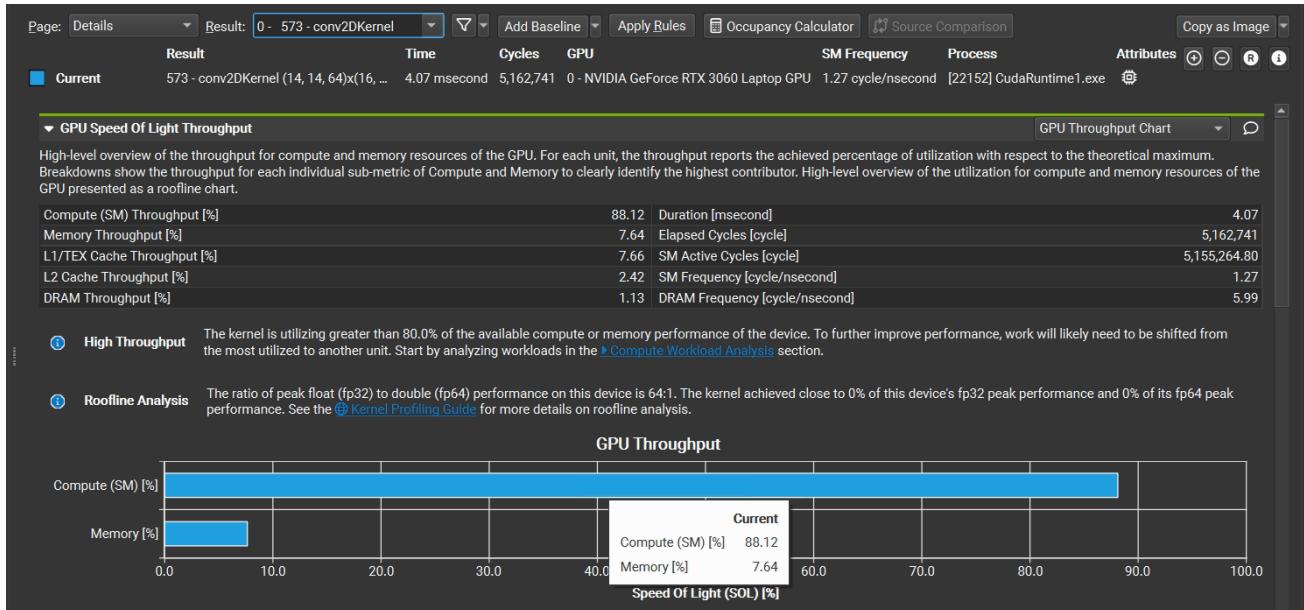




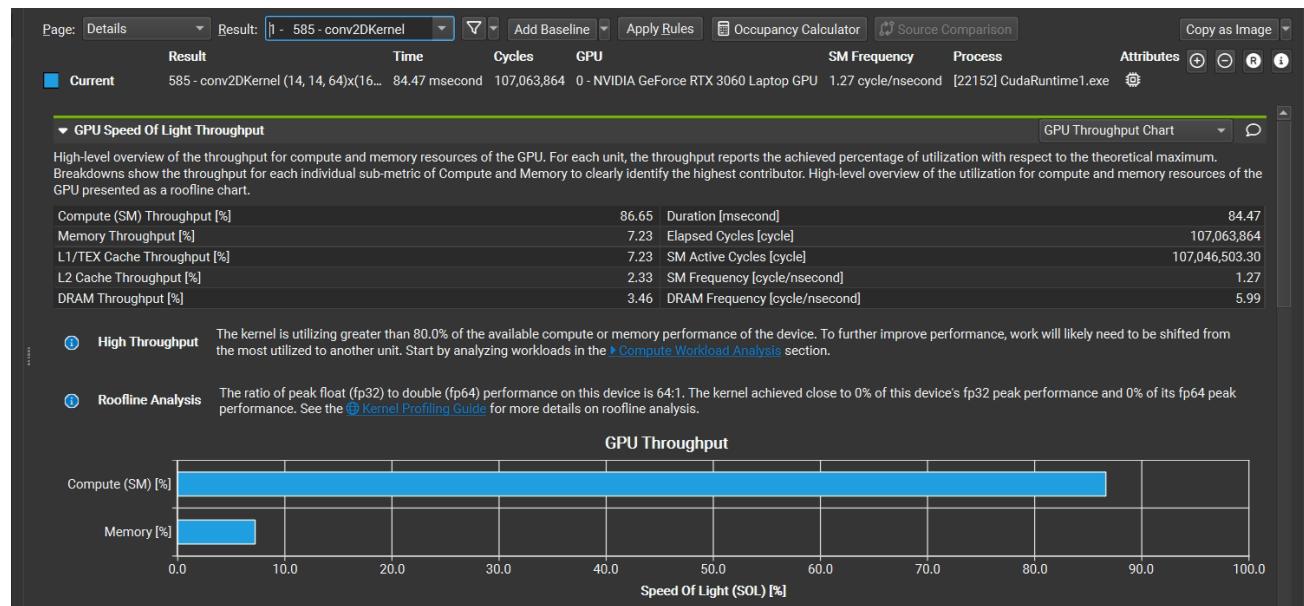


مشاهده می کنیم که بیشترین زمان اجرا مربوط به کرنل های کانولوشن هست و Memory Throughput کرنل های کانولوشن بیشتر است و Compute Throughput کرنل های کانولوشن.

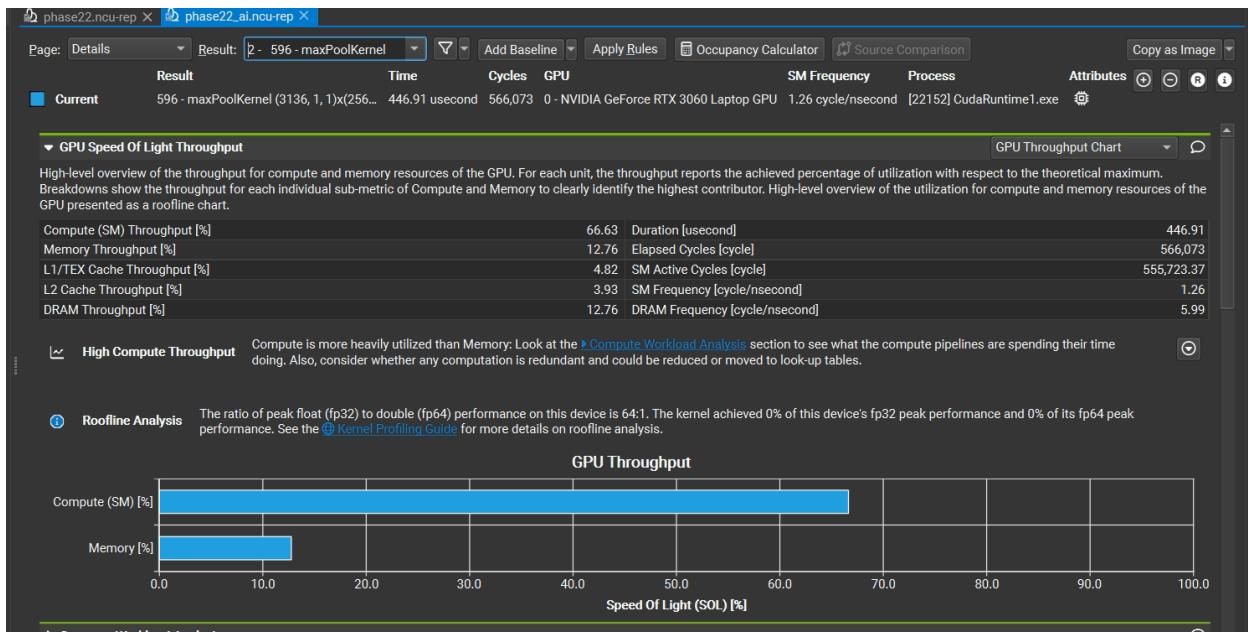
- Look at the GPU Speed Of Light Throughput tab. What is the compute and memory throughput of your kernel? Is it memory or compute bound?



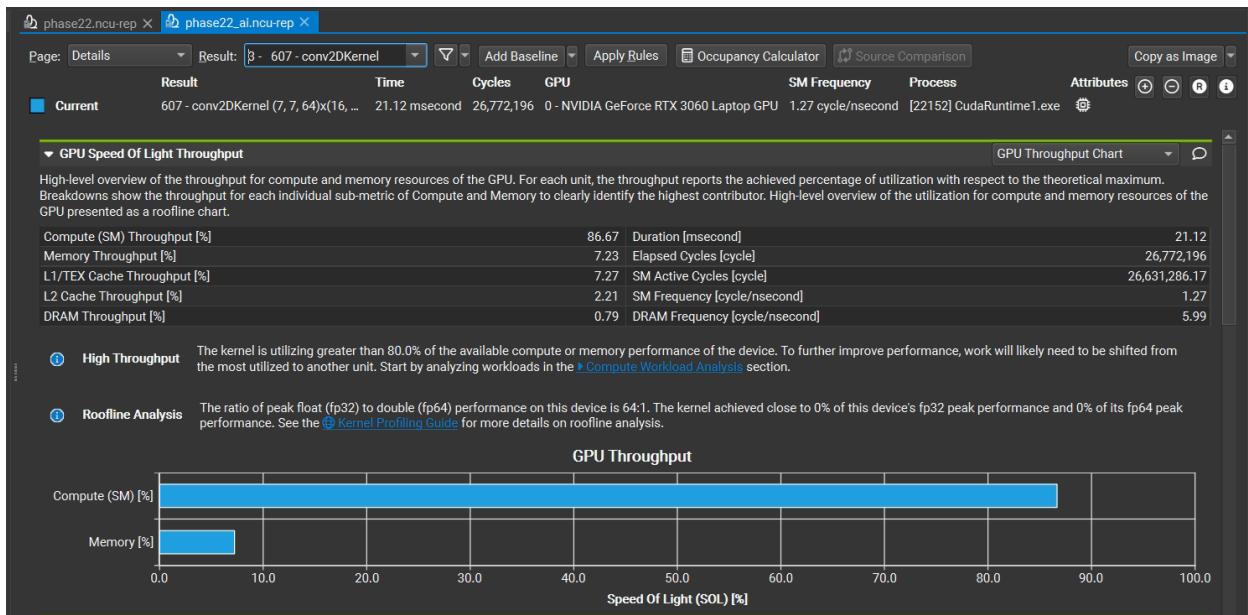
این کرنل **compute bound** است زیرا از ۸۸.۱۲ درصد از بخش **compute** استفاده کرده‌ایم در حالی که تنها ۷.۶ درصد از حافظه مصرف شده است و اگر توان محاسباتی بالاتری داشتیم می‌توانستیم پهنای باند بیشتری داشته باشیم.



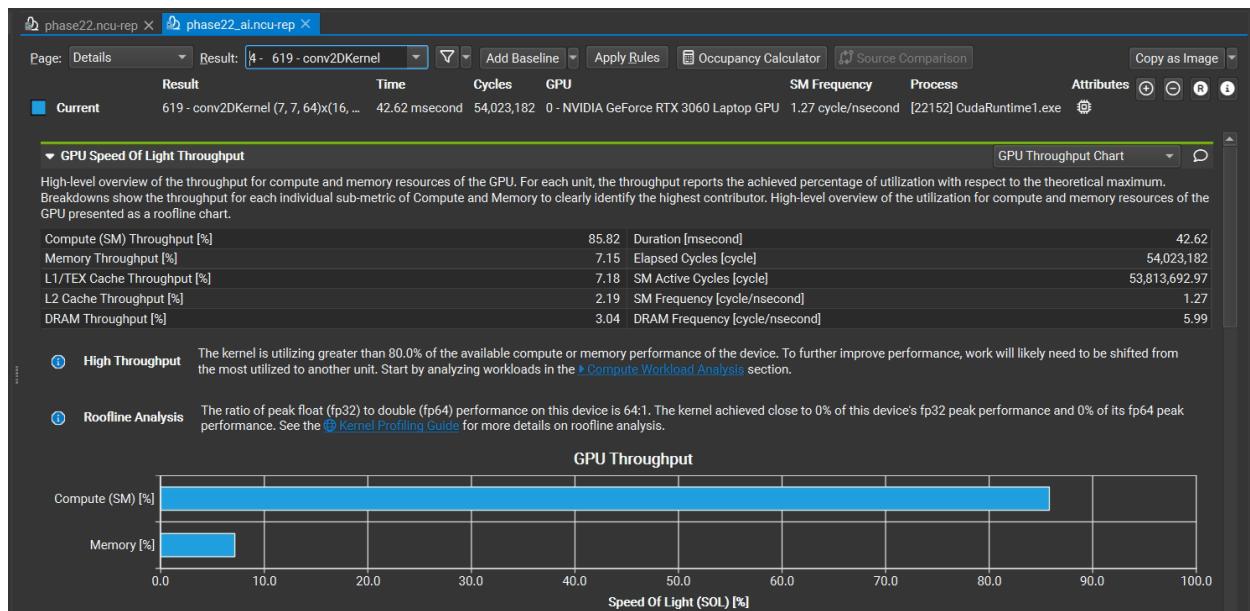
این کرنل **compute bound** است زیرا از ۸۶.۶۵ درصد از بخش **compute** استفاده کرده‌ایم در حالی که تنها ۷.۲۳ درصد از حافظه مصرف شده است و اگر توان محاسباتی بالاتری داشتیم می‌توانستیم پهنای باند بیشتری داشته باشیم.



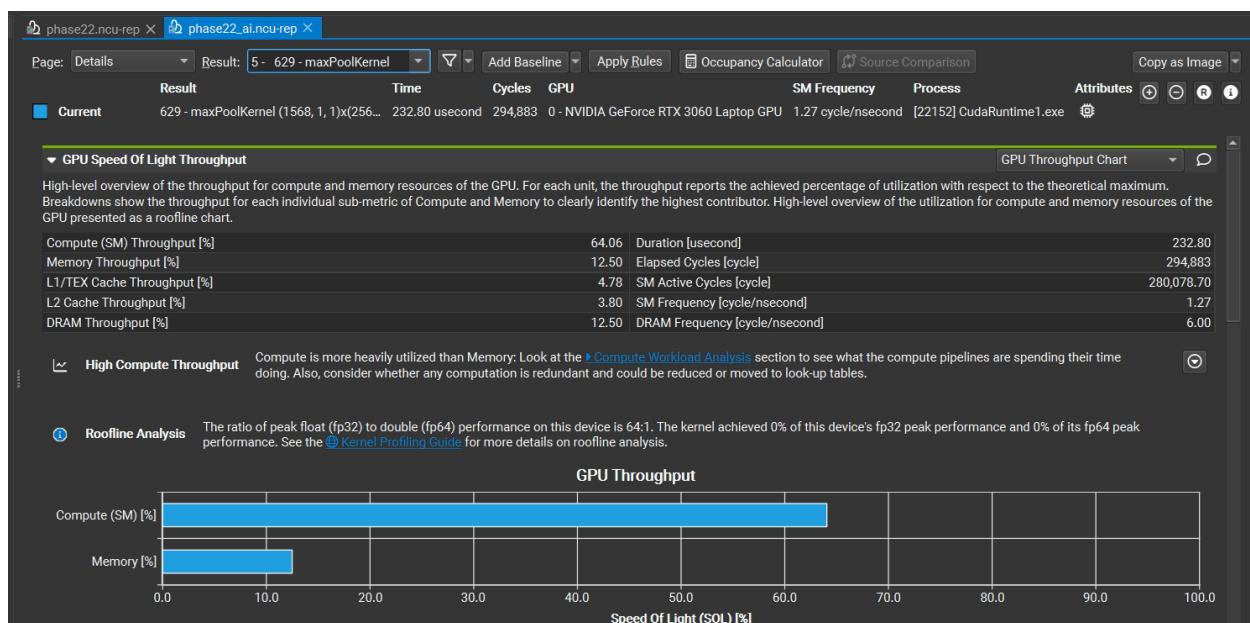
این کرنل **compute bound** است زیرا از ۶۶.۶۳ درصد از بخش **compute** استفاده کرده‌ایم در حالی که تنها ۱۲.۷۶ درصد از حافظه مصرف شده است و اگر توان محاسباتی بالاتری داشتیم می‌توانستیم پهنای باند بیشتری داشته باشیم.



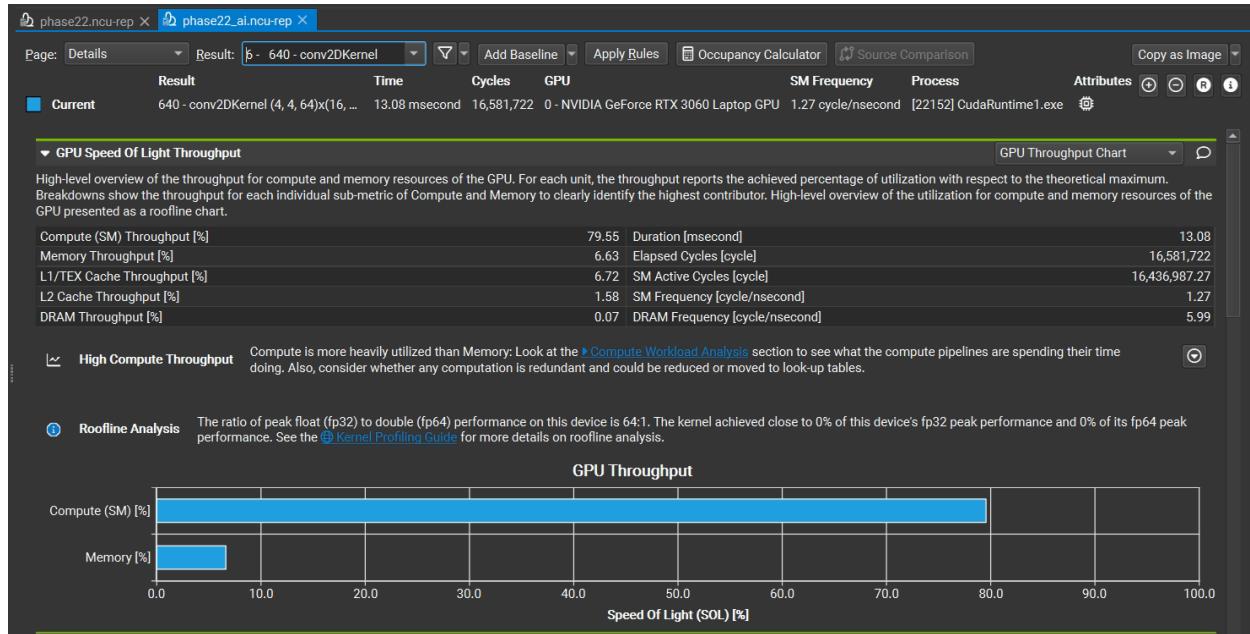
این کرنل **compute bound** است زیرا از ۸۶.۶۷ درصد از بخش **compute** استفاده کرده‌ایم در حالی که تنها ۷.۲۳ درصد از حافظه مصرف شده است و اگر توان محاسباتی بالاتری داشتیم می‌توانستیم پهنای باند بیشتری داشته باشیم.



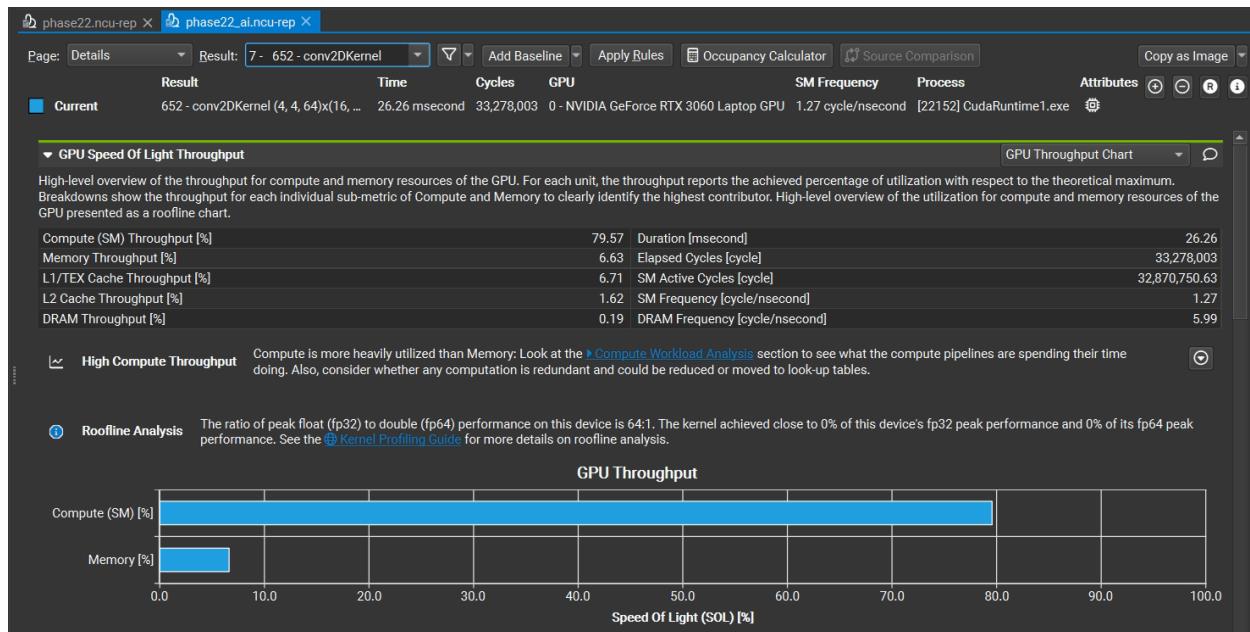
این کرنل **compute bound** است زیرا از ۸۵.۸۲ درصد از بخش **compute** استفاده کردایم در حالی که تنها ۷.۱۵ درصد از حافظه مصرف شده است و اگر توان محاسباتی، بالاتری داشتیم می‌توانستیم پنهانی باند بیشتری داشته باشیم.



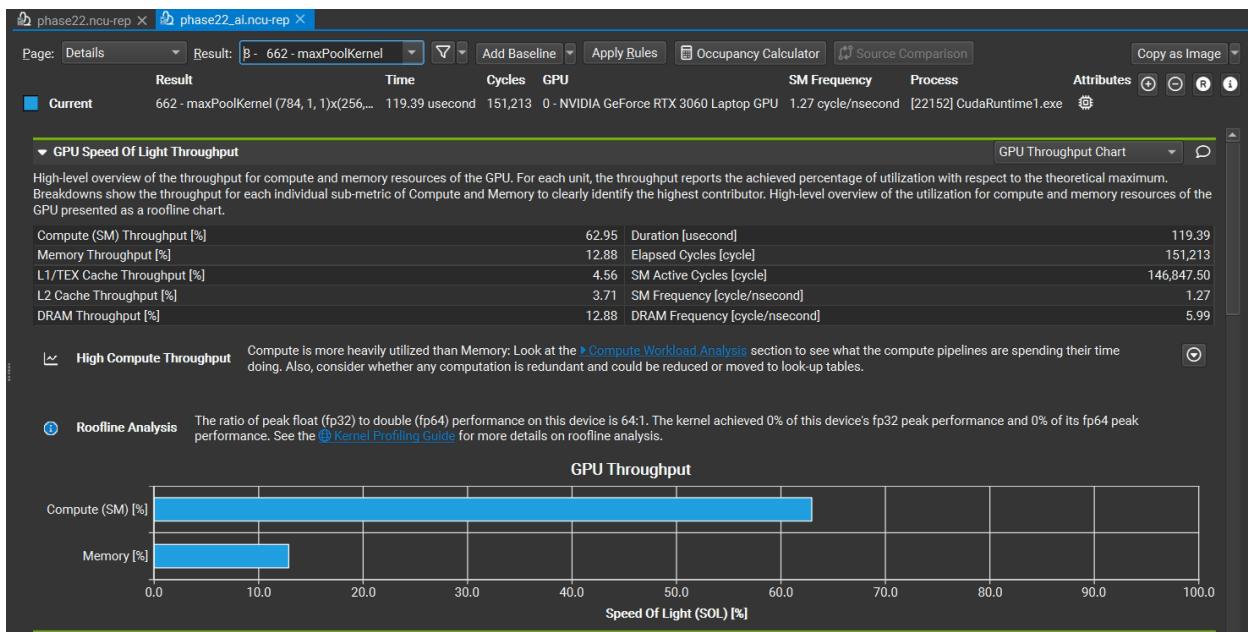
این کرنل **compute bound** است زیرا از ۶۴.۰۶ درصد از بخش **compute** استفاده کرده‌ایم در حالی که تنها ۱۲.۵۰ درصد از حافظه مصرف شده است و اگر توان محاسباتی بالاتری داشتیم می‌توانستیم بهترین باند پیشتری داشته باشیم.



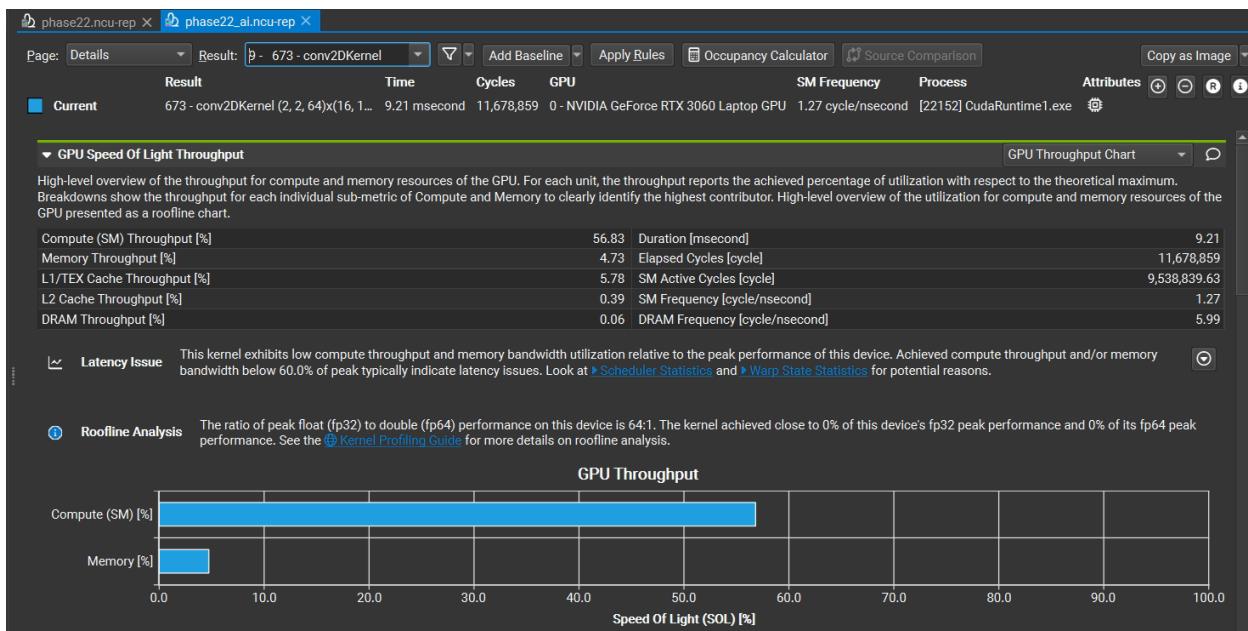
این کرنل **compute bound** است زیرا از ۷۹.۵۵ درصد از بخش **compute** استفاده کرده‌ایم در حالی که تنها ۶.۶۳ درصد از حافظه مصرف شده است و اگر توان محاسباتی بالاتری داشتیم می‌توانستیم پهنای باند بیشتری داشته باشیم.



این کرنل **compute bound** است زیرا از ۷۹.۵۷ درصد از بخش **compute** استفاده کرده‌ایم در حالی که تنها ۶.۶۳ درصد از حافظه مصرف شده است و اگر توان محاسباتی بالاتری داشتیم می‌توانستیم پهنای باند بیشتری داشته باشیم.

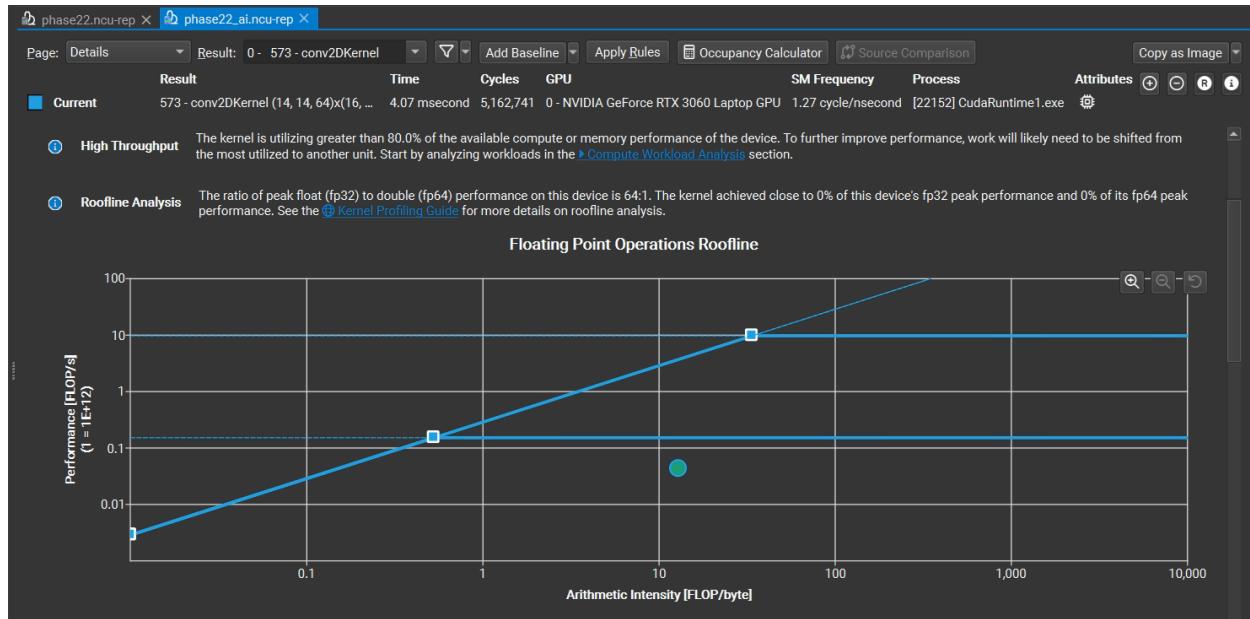


این کرنل **compute bound** است زیرا از ۶۲.۹۵ درصد از بخش **compute** استفاده کرده‌ایم در حالی که تنها ۱۲.۸۸ درصد از حافظه مصرف شده است و اگر توان محاسباتی بالاتری داشتیم می‌توانستیم پهنای باند بیشتری داشته باشیم.

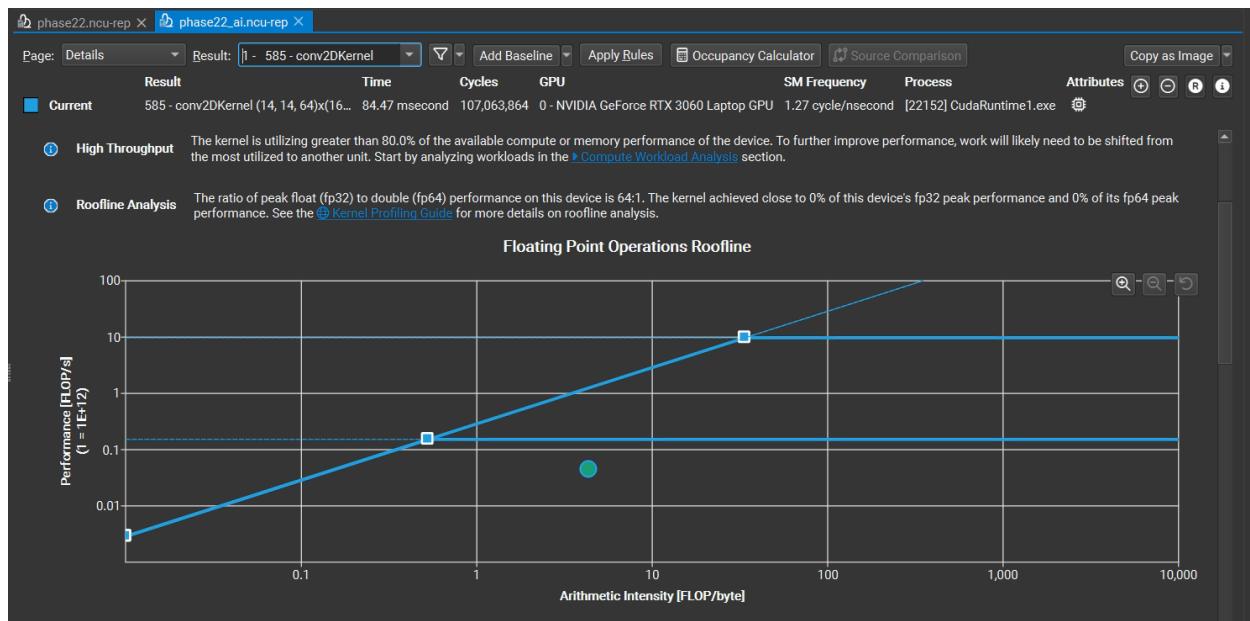


این کرنل **compute bound** است زیرا از ۵۶.۸۳ درصد از بخش **compute** استفاده کرده‌ایم در حالی که تنها ۴.۷۳ درصد از حافظه مصرف شده است و اگر توان محاسباتی بالاتری داشتیم می‌توانستیم پهنای باند بیشتری داشته باشیم.

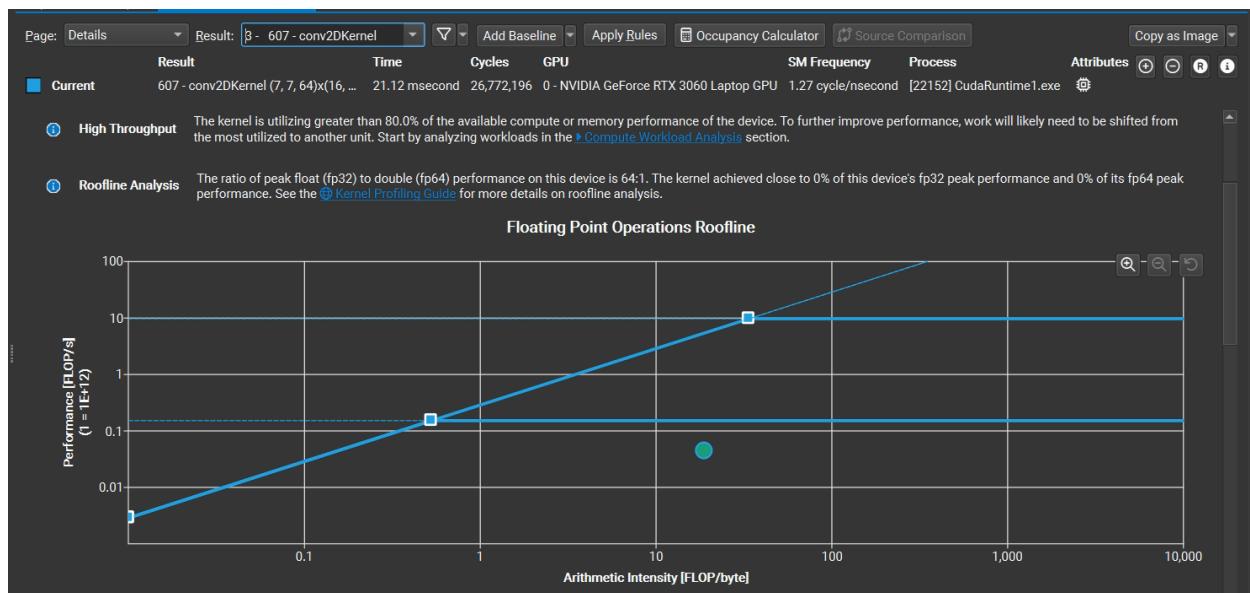
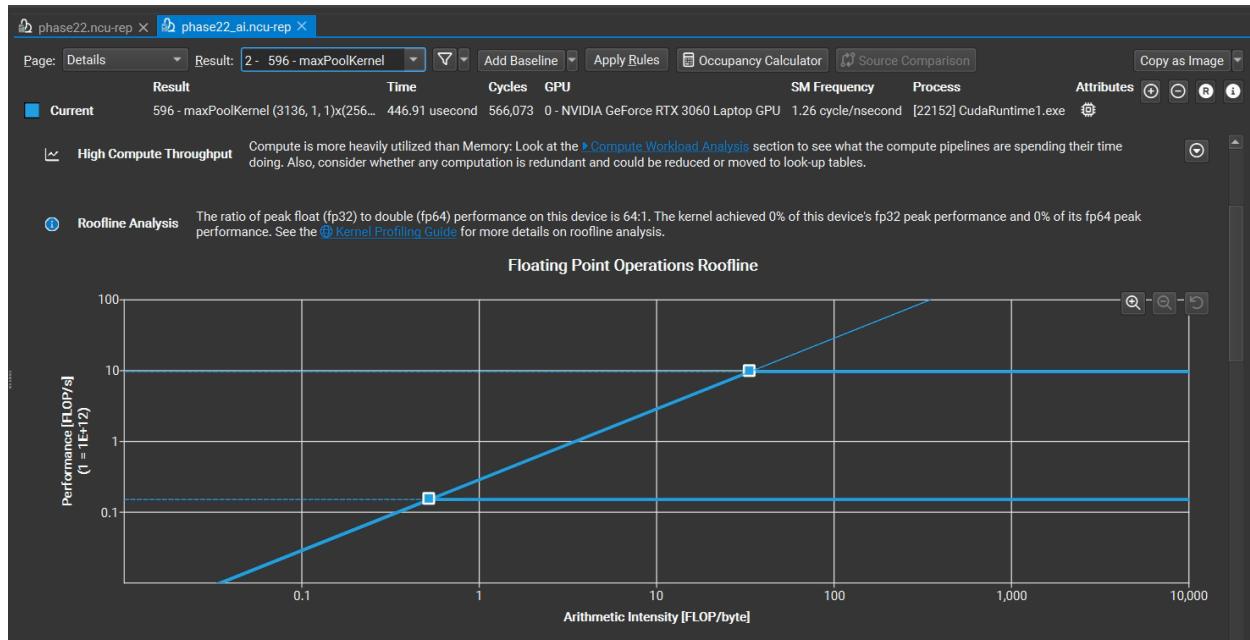
Look at the roofline graph of your kernel. Where in the graph does your kernel reside? Is there any room for improvement? Does the graph agree with your analysis of compute or memory bound in the last step?



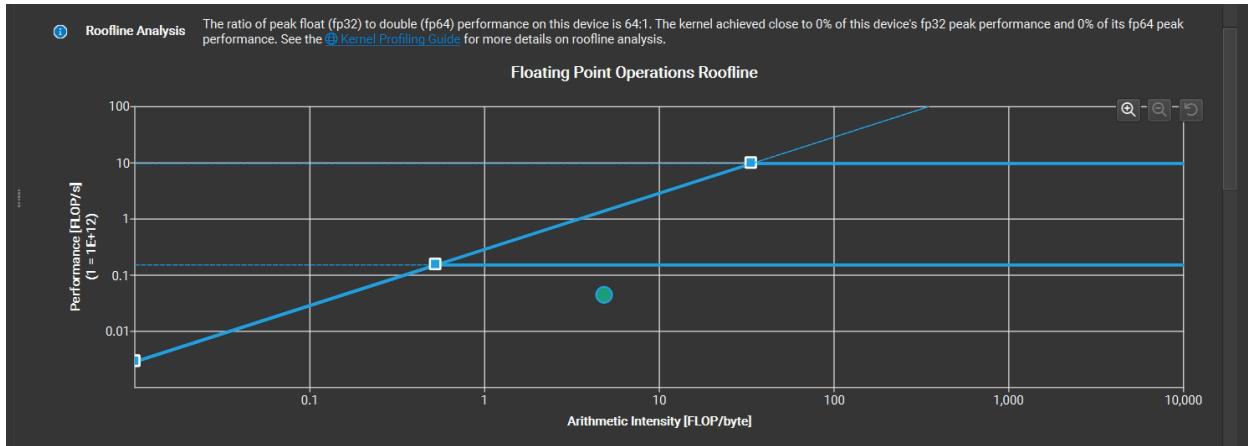
در محدوده compute bound هستیم.



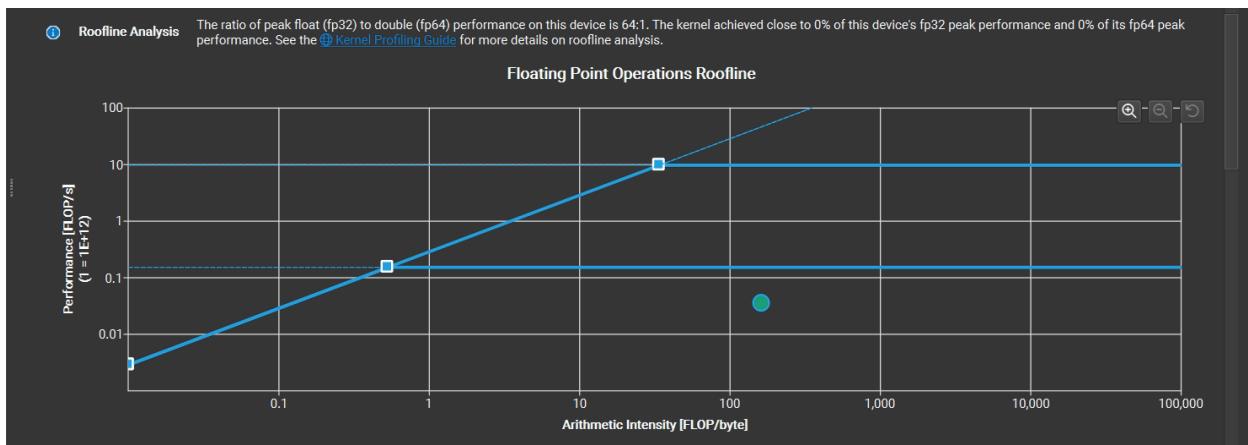
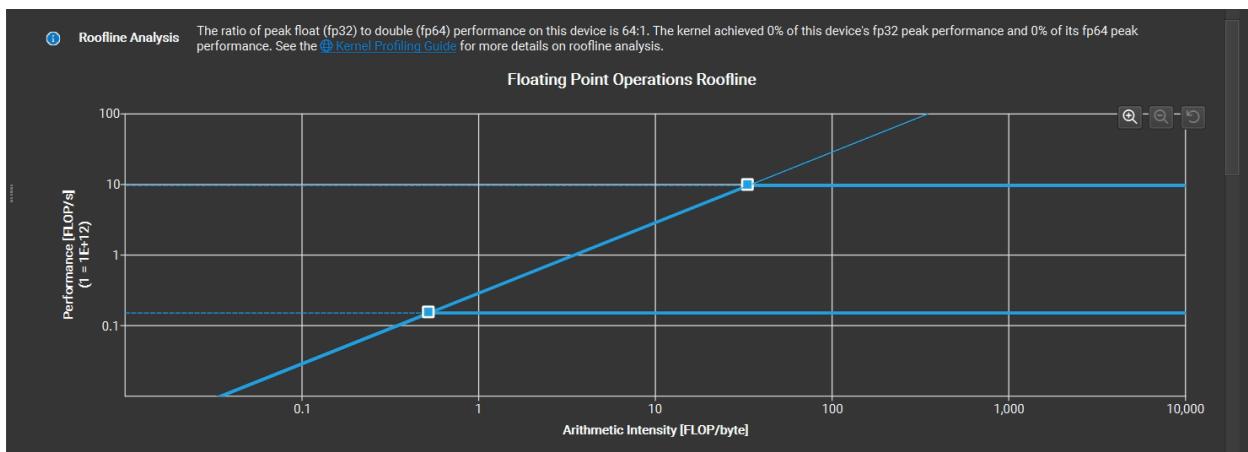
در محدوده compute bound هستیم.



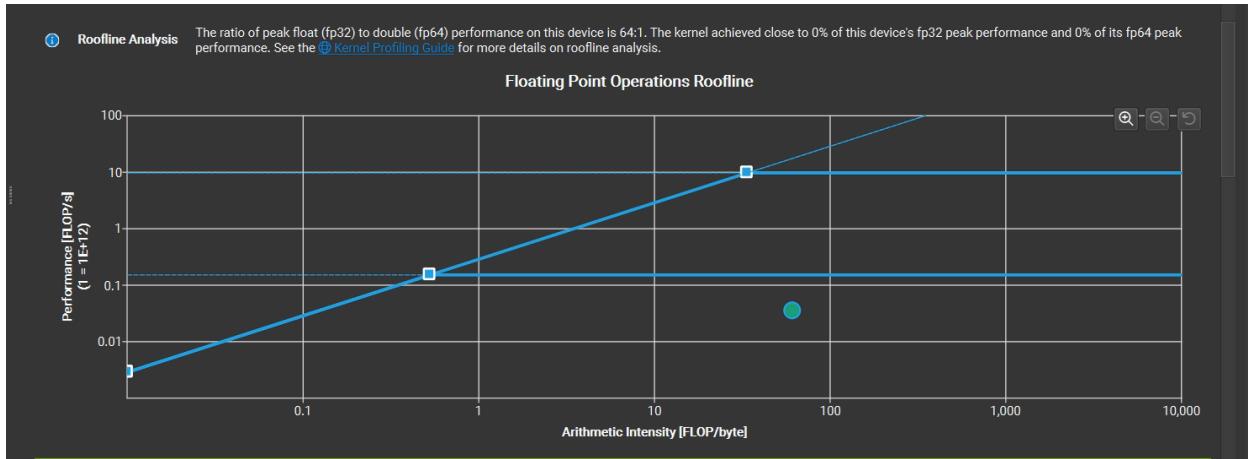
دراخونیم هستیم compute bound محدود داریم.



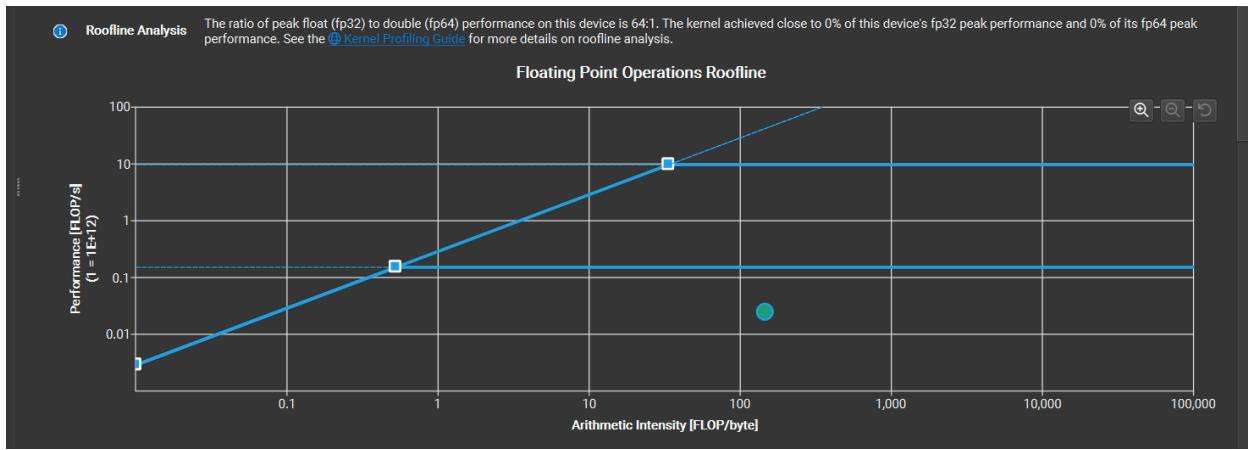
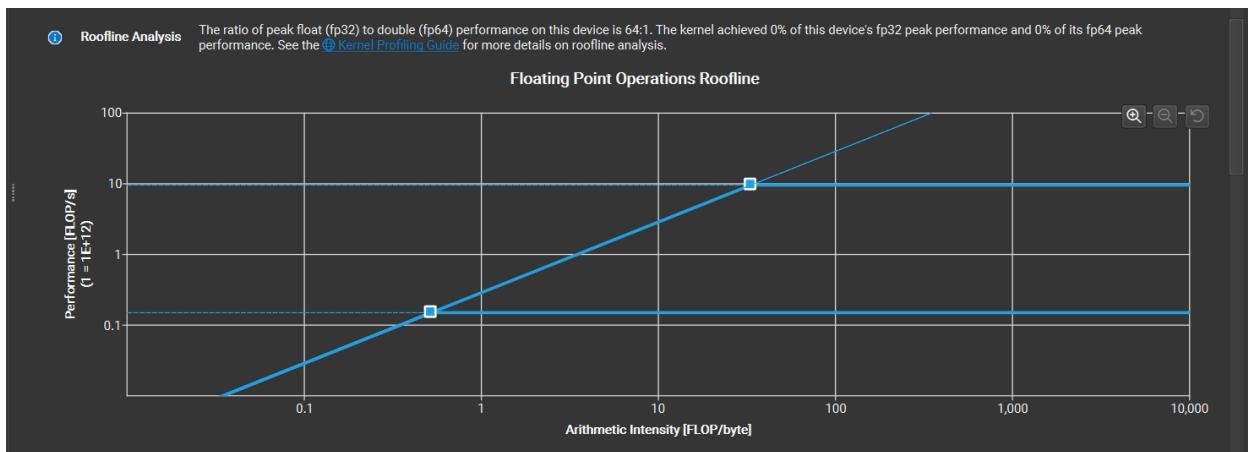
در محدوده compute bound هستیم.



در محدوده compute bound هستیم.



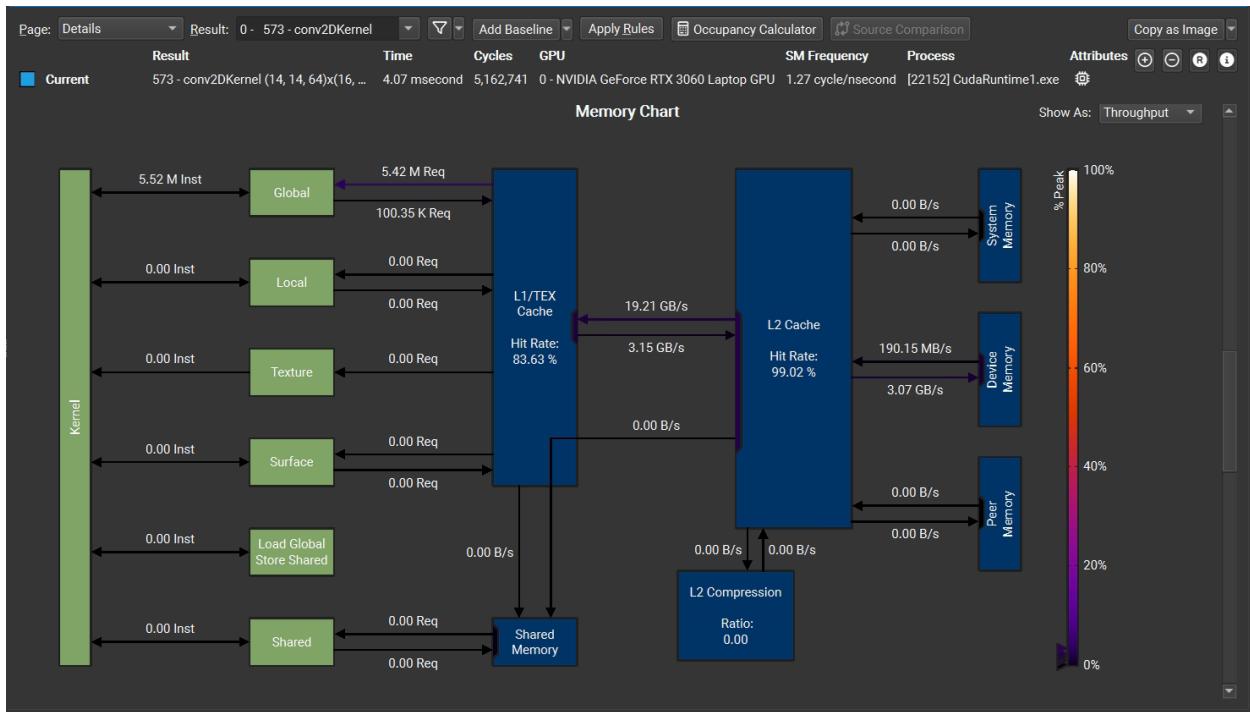
در محدوده compute bound هستیم.



در محدوده compute bound هستیم.

- Look at the memory workload analysis tab. Take note of the memory chart. What is the ratio of shared memory to global memory usage? What are your cache hit-rates? Does the tab give you any warning regarding uncoalesced memory accesses?

Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current 573 - conv2DKernel (14, 14, 64)x(16, ...)	4.07 msecound	5,162,741	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
Memory Throughput [Gbyte/second]	3.26	Mem Busy [%]	6.05			
L1/TEX Hit Rate [%]	83.63	Max Bandwidth [%]	7.64			
L2 Hit Rate [%]	99.02	Mem Pipes Busy [%]	16.58			
L2 Compression Success Rate [%]	0	L2 Compression Ratio	0			



Global Memory:

Requests: 5.42 million requests

Data transfer: 100.35K requests to L1/TEX Cache

Local, Texture, Surface, Shared Memory:

Usage:

این حافظه‌ها دستورات و درخواست‌های صفر را نشان می‌دهند که نشان می‌دهد در اجرای این هسته از آنها استفاده نشده است.

L1/TEX Cache:

Hit Rate: 83.63%

به این معنی که بخش قابل توجهی از درخواست های داده به طور مستقیم از این حافظه پنهان ارائه شده است.

Data transfer: 19.21 GB/s between L1/TEX Cache and L2 Cache.

L2 Cache:

Hit Rate: 99.02%

نشان دهنده استفاده بسیار کارآمد است.

Data transfer: 3.15 GB/s to L1/TEX Cache and 190.15 MB/s to Device Memory.

Device Memory:

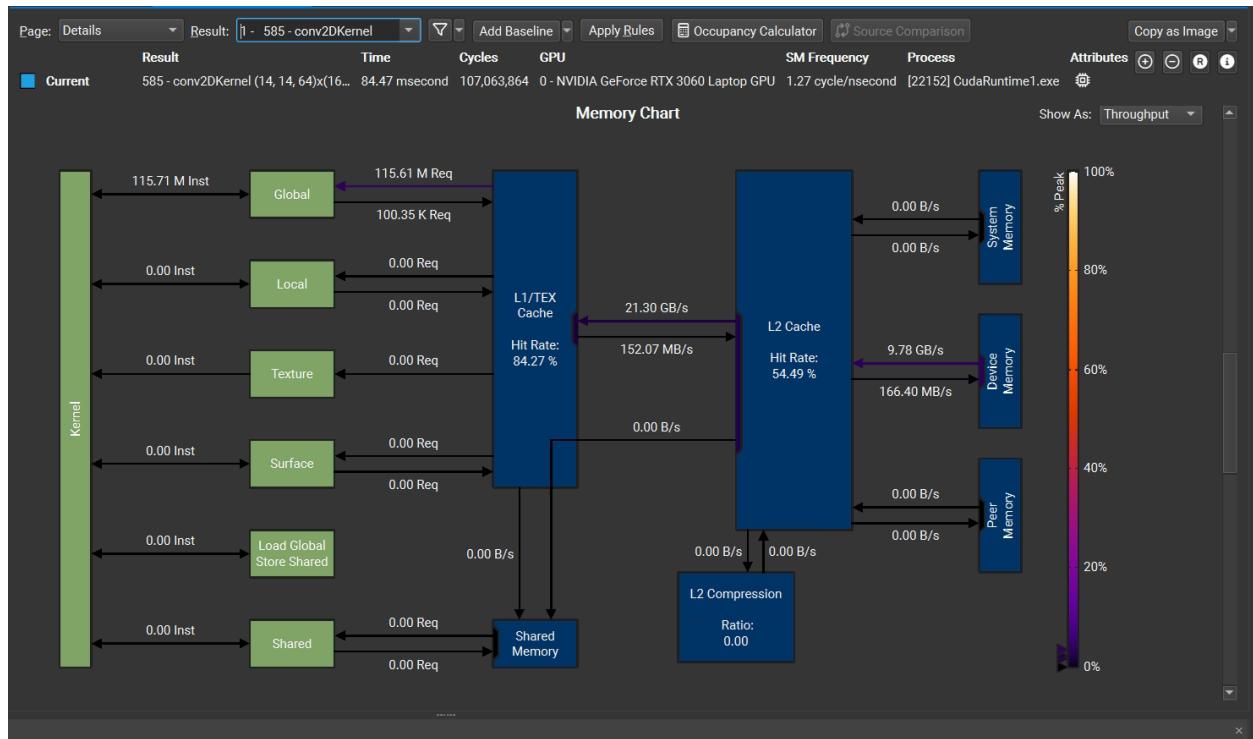
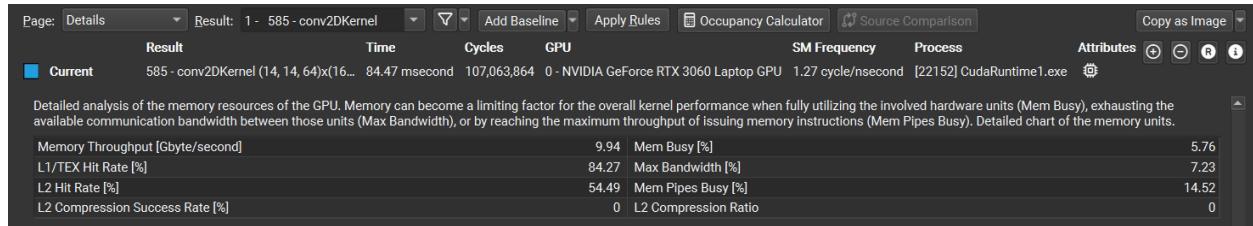
Data transfer: 3.07 GB/s to L2 Cache.

System Memory, Peer Memory, and Shared Memory:

استفاده: انتقال صفر بایت در ثانیه، نشان دهنده عدم حرکت داده در این بخشها است.

Performance Metrics

نوارهای توان عملیاتی در سمت راست نشان دهنده درصد حداکثر توان عملیاتی حافظه است که در بخش‌های مختلف سلسله مراتب حافظه به دست آمده است. نوار رنگ گرادیان نشان می‌دهد که استفاده از بنفس (کم) تا نارنجی (بالا) چقدر به حداکثر عملکرد نزدیک است.



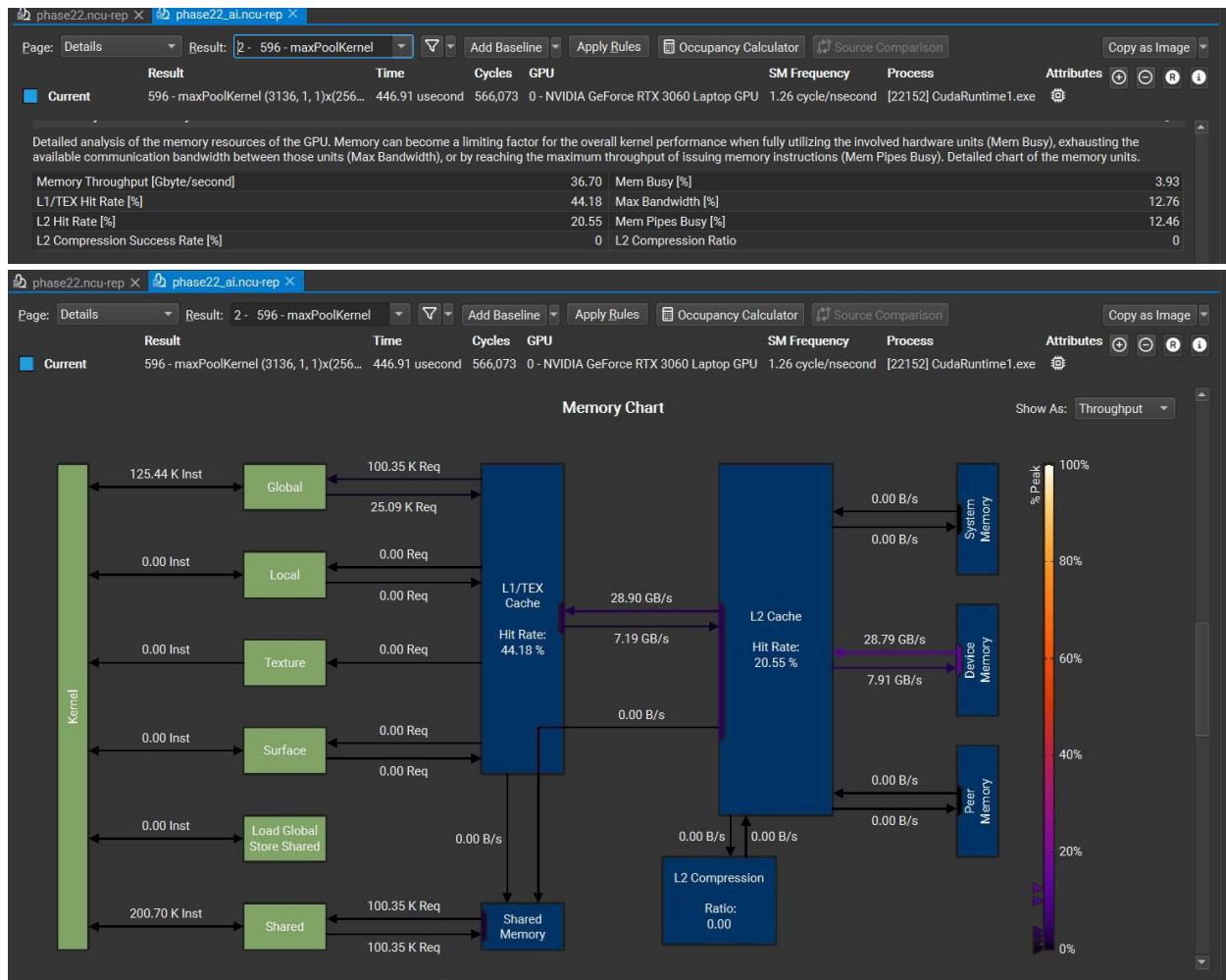
Shared to Global Memory Usage Ratio:

میزان استفاده از حافظه مشترک • کیلوبایت و میزان مصرف حافظه گلوبال ۱۱۵.۷۱ مگابایت است. از آنجایی که هیچ استفاده‌ای از حافظه مشترک وجود ندارد، نسبت ۱:۱۱۵.۷۱ خواهد بود، که به این معنی است که هیچ حافظه مشترکی در این زمینه استفاده نمی‌شود.

Cache Hit-Rates:

L1/Texture Cache Hit Rate: The hit rate for the L1/Texture Cache is 84.7%.

L2 Cache Hit Rate: The L2 Cache has a hit rate of 54.9%.



Shared to Global Memory Usage Ratio:

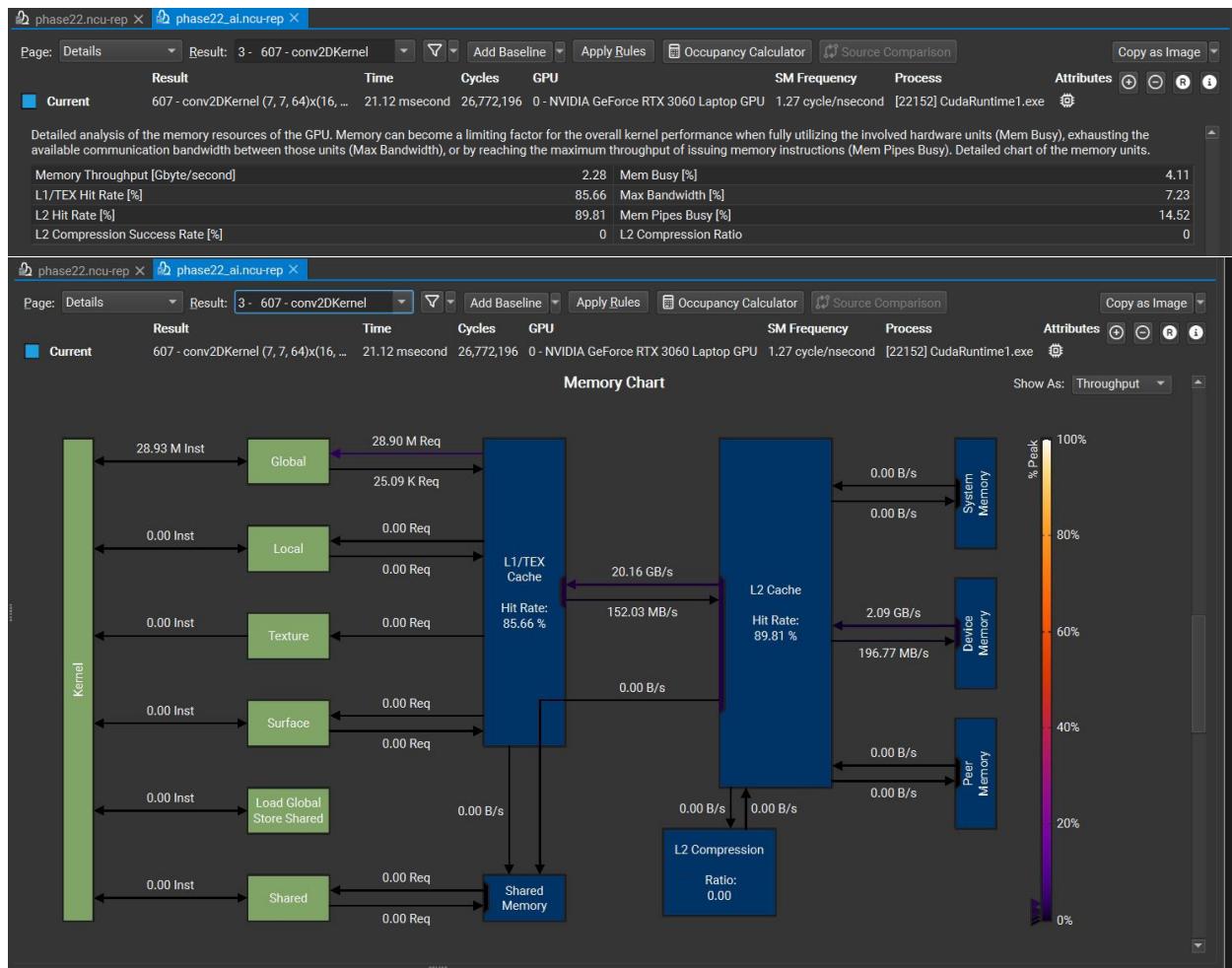
The shared memory usage is 20.704 KB, and the global memory usage is 10.352 KB. To calculate this ratio:

$$\text{Shared Memory}/\text{Global Memory} = 20.704 \text{ KB}/10.352 \text{ KB}$$

Cache Hit-Rates:

L1/TEX Cache Hit Rate: The hit rate for the L1/TEX Cache is 41.18%.

L2 Cache Hit Rate: The L2 Cache has a hit rate of 25.95%.



Shared to Global Memory Usage Ratio:

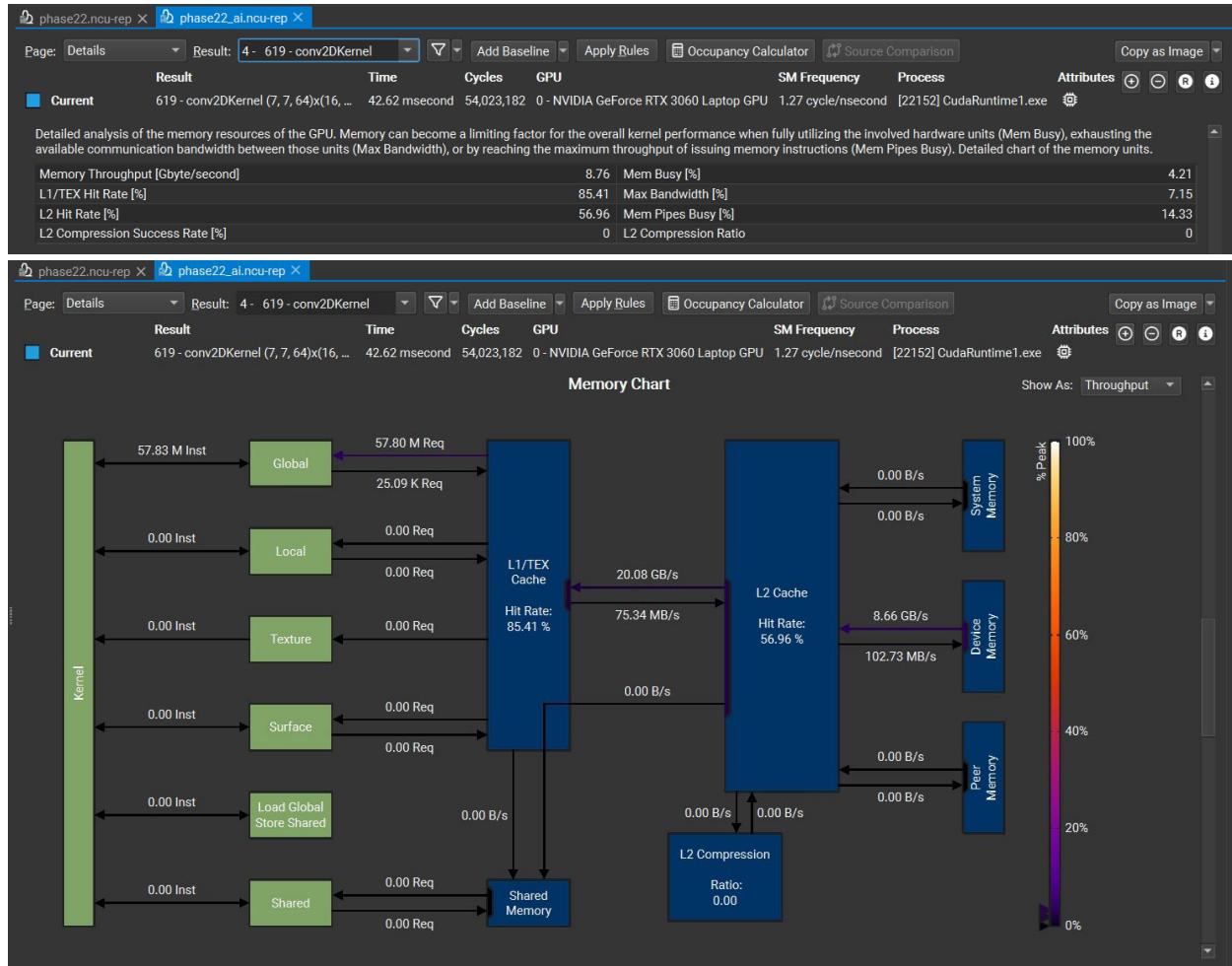
The shared memory is at 0.03 KB, and the global memory is at 2.50 KB. The ratio of shared to global memory usage is therefore:

$$0.03 \text{ KB} / 2.50 \text{ KB} \approx 1:83$$

Cache Hit-Rates:

L1/TEX Cache Hit Rate: 99.65%

L2 Cache Hit Rate: 98.31%



Shared to Global Memory Usage Ratio:

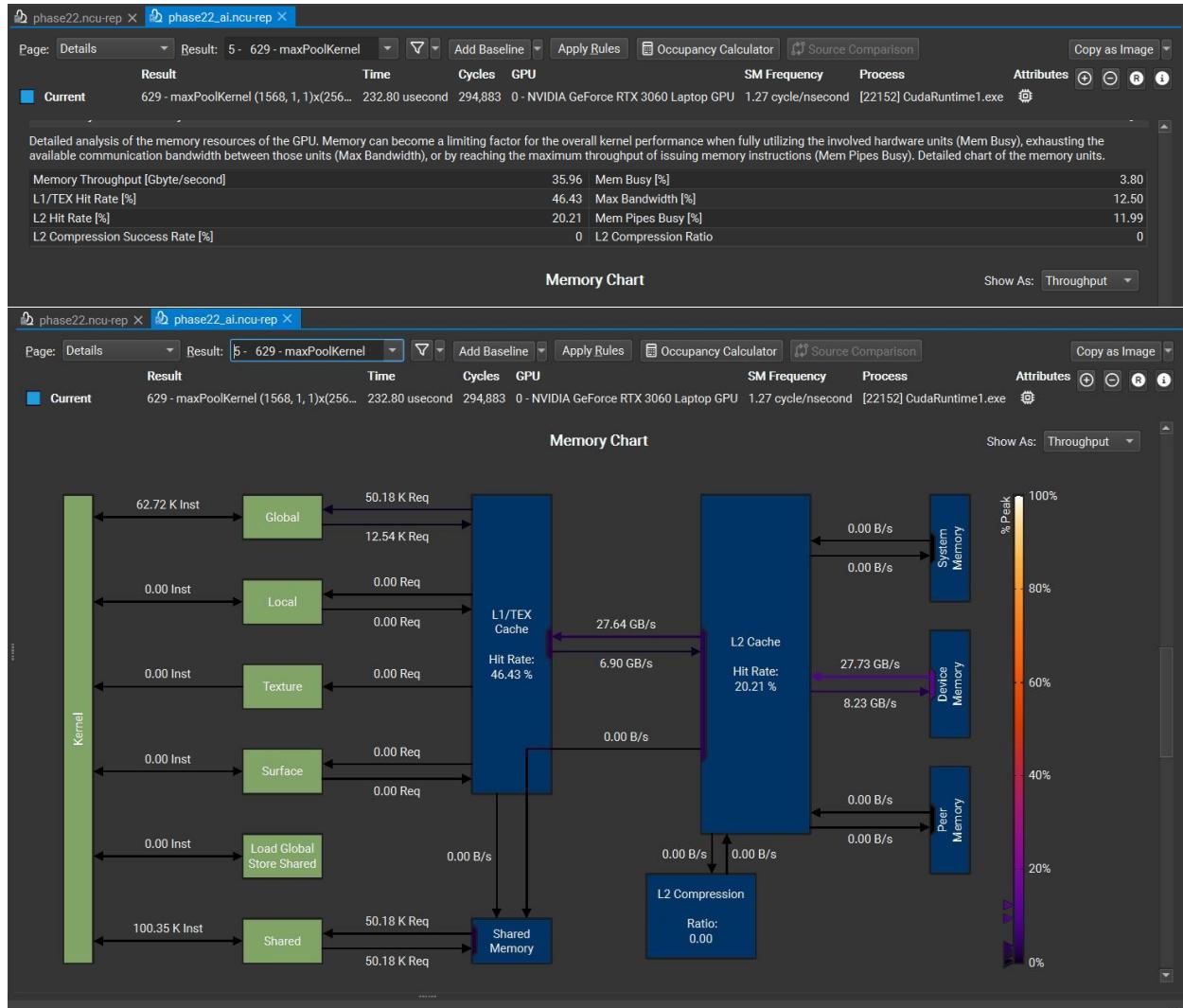
The shared memory usage is 0.02 MB, and the global memory usage is 57.03 MB. The ratio of shared to global memory usage is approximately:

$$0.02/57.03 \approx 0.00035$$

Cache Hit-Rates:

L1/TEX Cache Hit Rate: 25.9%

L2 Cache Hit Rate: 20.6%



Shared to Global Memory Usage Ratio:

The shared memory usage is 10.35KB, and the global memory usage is 58.31KB. The ratio of shared to global memory usage is approximately:

$$10.35/58.31 \approx 0.177 \text{ or about } 17.7\%$$

Cache Hit-Rates:

L1/Texture Cache Hit Rate: 99.6%

L2 Cache Hit Rate: 76%

- Look at the occupancy tab. What are the theoretical and achieved occupancy rates? Explain any possible discrepancy between the two.

Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
573 - conv2DKernel (14, 14, 64)x(16, ...)	4.07 msecound	5,162,741	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
warp(s) that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.						
Theoretical Occupancy [%]	100	Block Limit Registers [block]				6
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]				8
Achieved Occupancy [%]	97.85	Block Limit Warps [block]				6
Achieved Active Warps Per SM [warp]	46.97	Block Limit SM [block]				16
Page: Details Result: 1 - 585 - conv2DKernel	Add Baseline	Apply Rules	Occupancy Calculator	Source Comparison	Copy as Image	
Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
585 - conv2DKernel (14, 14, 64)x(16, ...)	84.47 msecound	107,063,864	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
warp(s) that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.						
Theoretical Occupancy [%]	100	Block Limit Registers [block]				6
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]				8
Achieved Occupancy [%]	96.98	Block Limit Warps [block]				6
Achieved Active Warps Per SM [warp]	46.55	Block Limit SM [block]				16
Page: Details Result: 2 - 596 - maxPoolKernel	Add Baseline	Apply Rules	Occupancy Calculator	Source Comparison	Copy as Image	
Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
596 - maxPoolKernel (3136, 1, 1)x(256, ...)	446.91 usecond	566,073	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.26 cycle/nsecond	[22152] CudaRuntime1.exe	
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warp(s) that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.						
Theoretical Occupancy [%]	66.67	Block Limit Registers [block]				4
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]				8
Achieved Occupancy [%]	65.13	Block Limit Warps [block]				6
Achieved Active Warps Per SM [warp]	31.26	Block Limit SM [block]				16
Page: Details Result: 3 - 607 - conv2DKernel	Add Baseline	Apply Rules	Occupancy Calculator	Source Comparison	Copy as Image	
Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
607 - conv2DKernel (7, 7, 64)x(16, ...)	21.12 msecound	26,772,196	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
warp(s) that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.						
Theoretical Occupancy [%]	100	Block Limit Registers [block]				6
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]				8
Achieved Occupancy [%]	96.74	Block Limit Warps [block]				6
Achieved Active Warps Per SM [warp]	46.44	Block Limit SM [block]				16
Page: Details Result: 4 - 619 - conv2DKernel	Add Baseline	Apply Rules	Occupancy Calculator	Source Comparison	Copy as Image	
Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
619 - conv2DKernel (7, 7, 64)x(16, ...)	42.62 msecond	54,023,182	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
warp(s) that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.						
Theoretical Occupancy [%]	100	Block Limit Registers [block]				6
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]				8
Achieved Occupancy [%]	96.61	Block Limit Warps [block]				6
Achieved Active Warps Per SM [warp]	46.37	Block Limit SM [block]				16
Page: Details Result: 5 - 629 - maxPoolKernel	Add Baseline	Apply Rules	Occupancy Calculator	Source Comparison	Copy as Image	
Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
629 - maxPoolKernel (1568, 1, 1)x(256, ...)	232.80 usecond	294,883	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warp(s) that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.						
Theoretical Occupancy [%]	66.67	Block Limit Registers [block]				4
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]				8
Achieved Occupancy [%]	65.04	Block Limit Warps [block]				6
Achieved Active Warps Per SM [warp]	31.22	Block Limit SM [block]				16

Result		Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	640 - conv2DKernel (4, 4, 64)x(16, ...	13.08 msecound	16,581,722	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.							
Theoretical Occupancy [%]	100	Block Limit Registers [block]	6				
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	8				
Achieved Occupancy [%]	86.38	Block Limit Warps [block]	6				
Achieved Active Warps Per SM [warp]	41.46	Block Limit SM [block]	16				
Result		Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	652 - conv2DKernel (4, 4, 64)x(16, ...	26.26 msecound	33,278,003	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.							
Theoretical Occupancy [%]	100	Block Limit Registers [block]	6				
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	8				
Achieved Occupancy [%]	86.44	Block Limit Warps [block]	6				
Achieved Active Warps Per SM [warp]	41.49	Block Limit SM [block]	16				
Result		Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	662 - maxPoolKernel (784, 1, 1)x(256, ...	119.39 usecond	151,213	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.							
Theoretical Occupancy [%]	66.67	Block Limit Registers [block]	4				
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	8				
Achieved Occupancy [%]	62.63	Block Limit Warps [block]	6				
Achieved Active Warps Per SM [warp]	30.06	Block Limit SM [block]	16				
phase22.ncu-rep phase22_ai.ncu-rep							
Result		Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	673 - conv2DKernel (2, 2, 64)x(16, 1, ...	9.21 msecond	11,678,859	0 - NVIDIA GeForce RTX 3060 Laptop GPU	1.27 cycle/nsecond	[22152] CudaRuntime1.exe	
warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.							
Theoretical Occupancy [%]	100	Block Limit Registers [block]	6				
Theoretical Active Warps per SM [warp]	48	Block Limit Shared Mem [block]	8				
Achieved Occupancy [%]	72.95	Block Limit Warps [block]	6				
Achieved Active Warps Per SM [warp]	35.01	Block Limit SM [block]	16				

نتایج هر ۱۰ بار فراخوانی کرنل‌ها با جزئیات آورده شده مثلاً برای کرنل اول داریم:

-Theoretical Occupancy: 100%

-Achieved Occupancy: 97.85%

CUDA در Occupancy به نسبت وارپ‌های فعال (گروه‌های رشته‌ها) به حداقل تعداد وارپ‌هایی که می‌توانند در یک SM در یک زمان فعال باشند اشاره دارد. Occupancy بالا به معنای استفاده کارآمد از منابع GPU است.

اختلاف بین این دو عدد را می‌توان به چند عامل نسبت داد:

۱. محدودیت‌های منابع: اگر کرنل از رجیسترها بیش از حد یا حافظه مشترک بیش از حد در هر بلوک استفاده کند، می‌تواند تعداد بلوک‌هایی را که می‌توانند در یک SM فعال باشند محدود کرده و در نتیجه مشغولیت به دست آمده را کاهش می‌دهد.

۲. انشعاب شاخه: اگر رشته‌های درون یک Warp مسیرهای اجرای متفاوتی را به دلیل عبارات شرطی دنبال کنند، می‌تواند منجر به بی‌کاری برخی از رشته‌ها شود و عملأً مشغولیت را کاهش دهد.

۳. تأخیر حافظه: مشغولیت زیاد می‌تواند با برنامه ریزی وارپ‌های دیگر در زمانی که فردمنتظر داده است، تأخیر حافظه را پنهان کند. با این حال، اگر الگوهای دسترسی به حافظه بهینه نشده باشند، مشغولیت به دست آمده ممکن است به افزایش عملکرد تبدیل نشود.

۴. ابعاد کرنل: ابعاد شبکه و بلوک تعریف شده هنگام راه اندازی کرنل ("blockSize, numBlocks") می‌تواند بر میزان مشغولیت تاثیر بگذارد. انتخاب‌های کمتر از حد مطلوب در اینجا می‌توانند از استفاده کامل از GPU جلوگیری کنند.

۵. محدودیت‌های سخت‌افزار: سخت‌افزار واقعی ممکن است دارای محدودیت‌هایی باشد که مانع از رسیدن آن به حداکثر تنوری می‌شود، مانند تعداد محدودی SM یا زمان‌بندی‌های تاب.

اختلافات زیاد بین مشغولیت نظری و مشغولیت به دست آمده معمولاً نشان‌دهنده بارهای کاری نامتعادل است که می‌تواند در نتیجه توزیع غیریکنواخت کار بین رشته‌ها یا بلوک‌ها باشد که منجر به کار بیش از حد برخی از آنها شود در حالی که برخی دیگر کم استفاده می‌مانند. توجه به این نکته مهم است که اگرچه مشغولیت بیشتر می‌تواند با ارائه فرصت‌های بیشتر برای پنهان کردن تأخیرها، عملکرد را بهبود بخشد، اما همیشه منجر به عملکرد بالاتر نمی‌شود. دستیابی به تعادل بین مشغولیت و سایر عوامل مانند پهنای باند حافظه و شدت محاسبات کلیدی برای بهینه سازی برنامه‌های CUDA است. هدف یافتن نقطه بهینه‌ای است که در آن منابع GPU به طور موثر و بدون ایجاد گلوگاه مورد استفاده قرار می‌گیرند.