SeamlessClone Project Overview

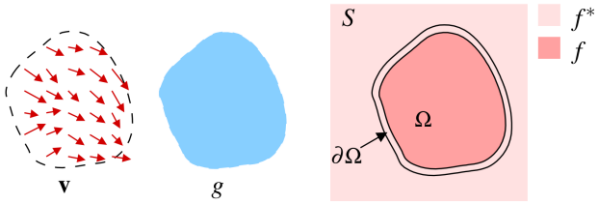
1.Project Overview

Seamless clone is an advanced image editing feature published in this paper, "Poisson Image Editing", OpenCV has this feature too. This project re-implements OpenCV seamless clone feature on NVIDIA GPU platform, using cuda programming.

1.1. Seamless Clone

Seamless clone is an advanced image editing feature, not like direct cloning image patch from source image to destination image, which has a drawback of inconsistent boundary colors. Seamless clone uses a guidance of vector in differential field and solve a Poisson equation with boundary pixels from destination image. This tech beautifully solves the problem of inconsistent drawback between source image and destination image. Details please refer to the original paper.

(a)



(b)

Fig. 1 Seamless clone concepts.

1.1. Project Status

Source path size	Max 592x592
OpenCV	3.4.5
Performance Latency	<30 ms
GPU	T4

Table 1, target setting.

This project is verified and tested with V100/T4 platform:

GPU driver	440.64
CUDA	10.2
OpenCV	3.4.5
OS	Ubuntu 18.04.4 LTS \n \l
GPU	V100/T4

Table 2, test platform configuration.

end2end performance after adding multiple CUDA streams support for FFT solver. Here we just not set the GPU clocks to maximum and leave it to application user.

Source Patch Size (WidthxHeight)	Performance (ms, 50 rounds with warm up) With/Without FFT		Device Memory (Bytes)	Error(1 in 255) With/Without FFT	
	V100	T4		Diff sum	Diff max
154x100	1.651/ 1.434	2.185 / 1.939	28165184	NA	
300x194	1.968 / 1.905	2.911 / 2.613	34728864	44/128	1/1
592x592	5.401 / 5.621	9.047 / 7.424	79377152	NA	
2400x1552	63.988/ 56.412	91.306 / 79.462	1070380192	17631/ 13659898	1/6

Tip: Use “nvidia-smi -i 0 -q -d CLOCK” to print the possible clock frequency configuration, and use something like “nvidia-smi -ac=5000,1590” to set top clock for memory and graphics. Then when you run the application, it will use this frequency for execution. Use “nvidia-smi -i 0 -rac” to reset the clock frequency.

Tested images:

(a) Destination image, 1600x898

(a) Output image, same size as destination image.

(c) Source image, 154x100, 300x194, 592x592

Fig. 2 Test images

2. Implementation Details

Please refer to "Poisson Image Editing" for tech background of seamless clone feature, and here for discrete Poisson equation solver for 2D images.

We refer the steps in section1 and section2 in here for our CUDA implementation. Remember to normalize matrix $V_{n1\text{-}1}$ and $V_{n2\text{-}1}$ before using it, and set h_x and h_y to 1. We didn't use cuFFT for faster DST implementation, instead we use cuBLAS which is much straight forward; if necessary, give cuFFT a try, check section **Error! Reference source not found.** for some tech background description.

The steps involved in our seamless clone implementation are:

1. Initialization mainly allocates device memory.
2. Init mask. To align exactly what OpenCV did in pre-processing, we input test code in OpenCV source code. We also developed an erode and a boundary box calculation operator using CUDA kernels. The modified mask is smaller than original, according to how user design it.

❖ Erode operator;	CUDA kernel
❖ Bounding box operator;	CUDA kernel

3. Calculate Laplacian filter for destination/source images. We use this equation to perform Poisson operator for each pixel in RGBP format:

$$(u_{i,j-1}-2u_{i,j}+u_{i,j+1})/h_x^2 + (u_{i-1,j}-2u_{i,j}+u_{i+1,j})/h_y^2 = F(x_i,y_j)$$

For each pixel $u_{i,j}$, { $u_{i,j-1}$, $u_{i,j+1}$, $u_{i-1,j}$, $u_{i+1,j}$ } are its four neighbor pixels. After Poisson filter, we have F_r^s, F_g^s, F_b^s Poisson operators for source images, and F_r^d, F_g^d, F_b^d for destination image respectively. We use two CUDA kernels for this calculation, one kernel for gradient in X/Y axis for destination/source images and output *gdX*/*gdY*; followed by another gradient filter which read in *gdX*/*gdY* and output Laplacian operator for each pixel, in format RGBP format, stored in *g*. The boundary condition is handled same as implemented in OpenCV3.4.5, using BORDER_DEFAULT. Destination/source images' *gdX*/*gdY* is also blended using mask calculated form step1.

❖ Calculate gradient in X/Y axis;	CUDA kernel
❖ Calculate another gradient filter, output Laplacian operation;	CUDA kernel

4. Solve Poisson equation. We refer to this equation to calculate Poisson equation using CUDA:

❖ Apply $V_{n2\text{-}1}$ to the rows of *g;*	cuBLAS
❖ Apply $V_{n1\text{-}1}^{-1}$ to the columns of *g;*	cuBLAS
❖ Calculate u~ using the relationship: $u\tilde{}_{i,j} = g\tilde{}_{i,j} / (\text{lambda}^i_{n1\text{-}1} + (h^2 \text{lambda}^j_{n2\text{-}1}))$;	CUDA kernel
❖ Apply $V_{n2\text{-}1}^{-1}$ to the rows of *u;*	cuBLAS
❖ Apply $V_{n1\text{-}1}$ to the columns of *u;*	cuBLAS

According to here, $V_{n2\text{-}1}$ is identical to its inverse $V_{n2\text{-}1}^{-1}$, same for $V_{n1\text{-}1}$, and no computing power is necessary to calculate it.

2.1. Optimization Tips

2.1.1. Initialization

We group all the device memory allocation operations in initialization step, and free all the allocated device memory just before application exit. So there is no need to break the GPU computation pipeline for pre-processing and Poisson solver.

2.1.2. Minimal Memory Copy

OpenCV implementation has many tiny operators while data pre-processing, like sub image crop, int to float type conversion, etc.; which are much a memory bound problem, slow down the overall performance. In our first version implementation, we use NPP for data pre-processing, which has the same problem. At last, while performance optimization, we remove all these memory copy operations into one kernel, avoid unnecessary read/write of device memory.

2.1.3. CUDA Kernel Merging

Small operations are all grouped into one CUDA kernel as much as possible, which avoid unnecessary read/write of device memory and kernel launch operations.

2.1.4. CUDA Stream

We use one CUDA stream for all the data pre-processing and Poisson equation solver, which make it possible for potential performance improvement using multiple threads. This strategy avoids unnecessary CUDA stream synchronize operations.

2.1.5. Warm Up and Multiple Run

Warm up and multiple run is necessary for true performance timing, like this pseudo code shows:

```
Func();
Timing(begin);
For( int loop=0; loop<Loops; loop++ ){
```

```
        Func();
        Synchronize(CUDA stream);
}
Timing(end);
Print_time(end-start);
```

2.1.6. Nvprof Analysis

Here shows one snapshot of the performance for one round (in 50) of computing. As you can see, there are two CUDA streams used here, which is not under our plan. Using multiple CUDA streams under one pipeline involves unnecessary stream synchronize operations, which breaks the pipeline into several parts and do harm the overall performance. We remove usage of stream 0 by specific all the operations to one specific CUDA stream, here is stream 21.

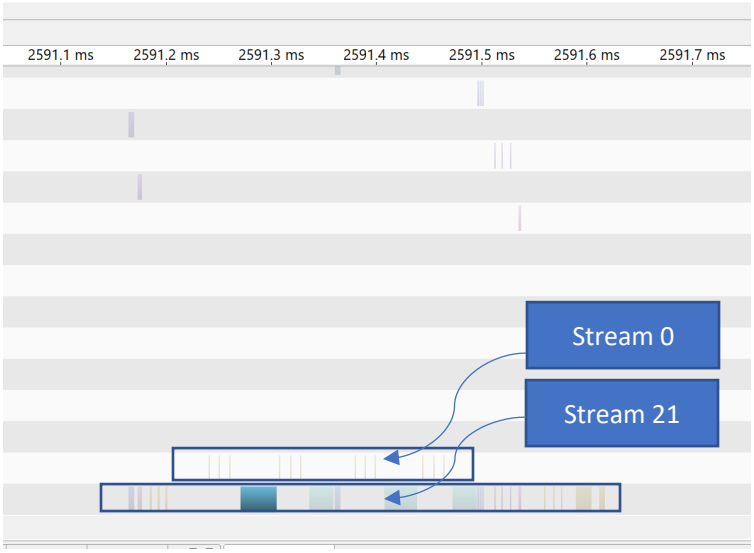


Fig. 3 NVVP performance timeline

2.1.7. Remove Tiny Malloc & Memcpy

After step 2.1.6, the performance timeline looks like this, there is no stream 0 usage anymore. But we find some tiny malloc and memcpy operations before each GEMM, which allocates tiny device memory and copy pptrA/pptrB/pptrC from host to device, this malloc operations breaks pipeline of CUDA stream execution too. We move this parts to initialization stage decrease latency to 0.678ms.

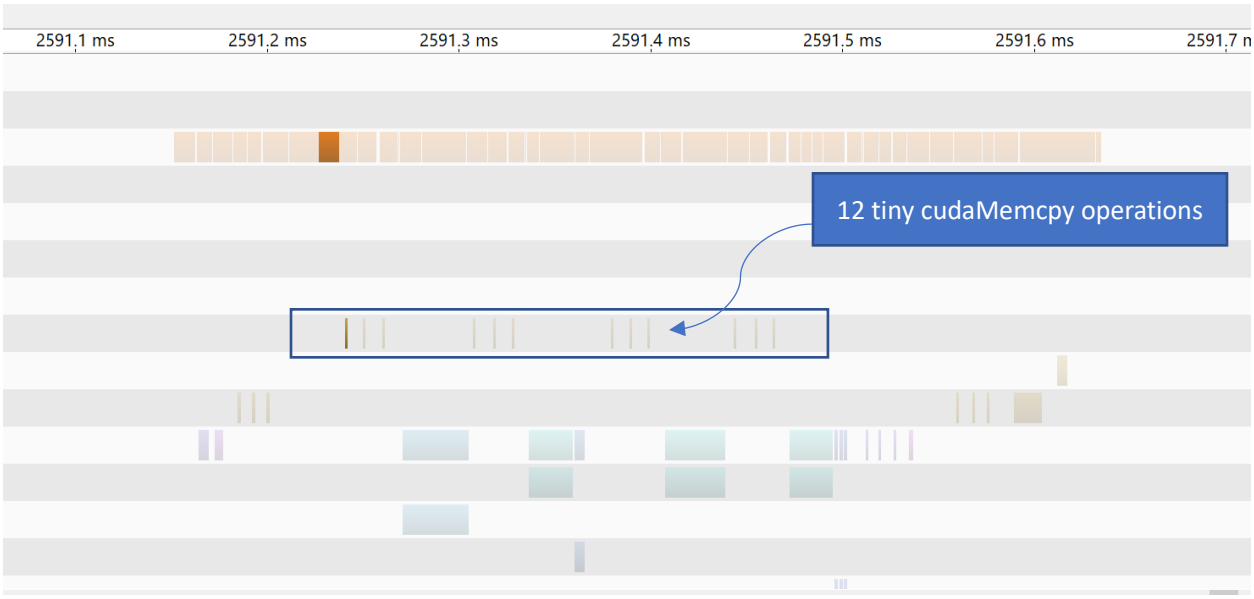


Fig. 4 NVVP performance timeline, before remove tiny malloc & memcpy

Code scripts for reference:

```
float* pptrA[3]    = { NULL, NULL, NULL };
float* pptrB[3]    = { NULL, NULL, NULL };
float* pptrC[3]    = { NULL, NULL, NULL };
for( int ch = 0; ch<C.mChannel; ch++ )
{
        pptrA[ch] = (A.mChannel==3)?(float*)A.ptr(0, 0, ch):(float*)A.ptr(0, 0, 0);
        pptrB[ch] = (B.mChannel==3)?(float*)B.ptr(0, 0, ch):(float*)B.ptr(0, 0, 0);
        pptrC[ch] = (float*)C.ptr(0, 0, ch);
}
float** dpptrA;
float** dpptrB;
float** dpptrC;
checkCudaErrors(cudaMalloc((void **)&dpptrA, C.mChannel * sizeof(*dpptrA)));
checkCudaErrors(cudaMalloc((void **)&dpptrB, C.mChannel * sizeof(*dpptrB)));
checkCudaErrors(cudaMalloc((void **)&dpptrC, C.mChannel * sizeof(*dpptrC)));
checkCudaErrors( cudaMemcpy(dpptrA, pptrA, sizeof(*dpptrA)*C.mChannel, cudaMemcpyHostToDevice) );
checkCudaErrors( cudaMemcpy(dpptrB, pptrB, sizeof(*dpptrB)*C.mChannel, cudaMemcpyHostToDevice) );
checkCudaErrors( cudaMemcpy(dpptrC, pptrC, sizeof(*dpptrC)*C.mChannel, cudaMemcpyHostToDevice) );
cublasSgemmBatched( mCublasHandle, transa, CUBLAS_OP_N, M, N, K, &alpha, dpptrA,
                    transa==CUBLAS_OP_T?K:M, dpptrB, K, &beta, dpptrC, M, C.mChannel );
```

0.678ms

2.1.8. Merge Kernels

After this, the performance timeline looks like this, there is no tiny memory allocation and copy operations

between GEMMs. Next, to remove memory copy after Laplacian operator, we move this operation to the end of Laplacian operator, avoid unnecessary memory copy.

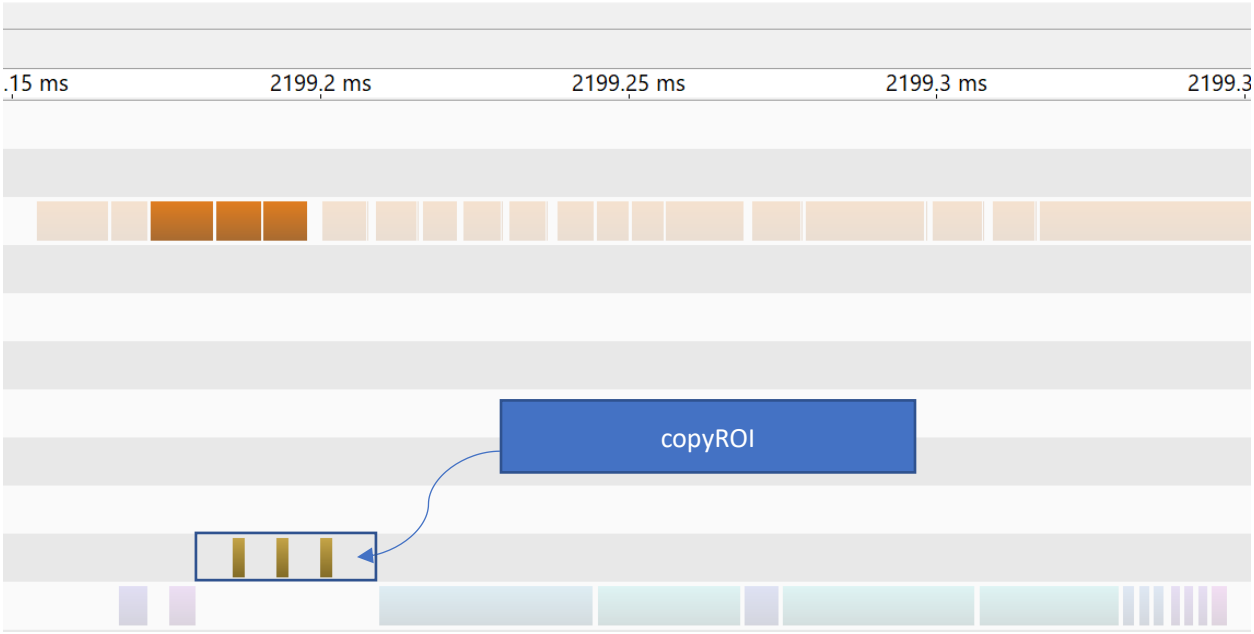


Fig. 5 NVVP performance timeline, before merge kernels

Code scripts for reference:

```
pre_process_v2(); // Laplacian operator
myNpp.copyROI( g, 0, 0, lapXY, 1, 1, g.mWidth, g.mHeight, mStream ); // move this copy to end of Laplacian operator.
```

0.563ms

2.1.9. Merge Post-processing Operations

After step 2.1.8, NVVP performance timeline looks this this, there is no memory copy for copyROI exists. Next we need to merge all the post-processing operations into one kernel. It involves operations transpose, elementwise scale, crop to [0,255], convert to unsigned char and copyROI to output image with specified offsetXY; these all are elementwise operations. Before this optimization, we implement post-processing with NPP for convenience. But for best performance, we need to merge elementwise operations as much as possible.

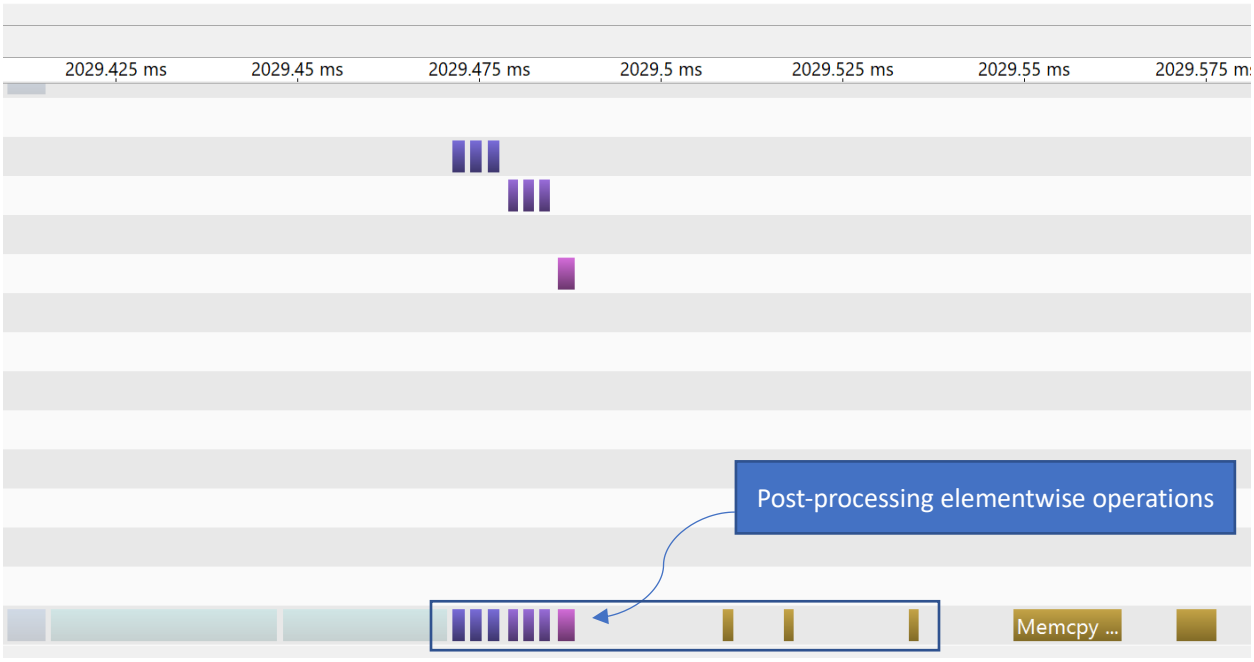


Fig. 6 NVVP performance timeline, before merge post-processing operations

Code scripts for reference:

```
myNpp.transpose( &u, &g );
myNpp.convertFloat2UC( &u_, &u, 2.0f );
myNpp.copyROI( ucRGB_Output, leftTop.x+1, leftTop.y+1,u_, 0, 0, u_.mWidth, u_.mHeight,mStream );
```

0.474ms

2.1.10. Merge Post-processing Operations

After step 2.1.9, all post-processing operations are merged into one CUDA kernel. Removing unneeded copy to host operation, the performance timeline looks much better than before.

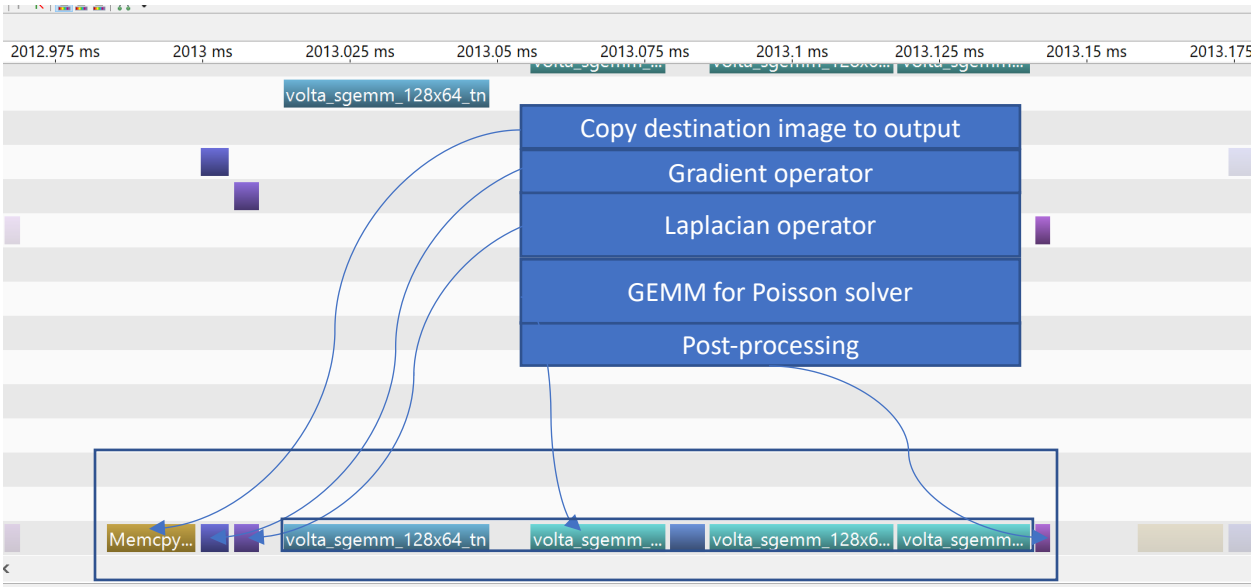


Fig. 7 NVVP performance timeline, after merging post-processing operations into one CUDA kernel.

2.1.11. Cooperative Groups

After step 2.1.10, we did kernel merging with cooperative groups for 3 erode operations at initialization stage, 2 CUDA kernels at pre-processing stage while calculating Laplacian operator. Now each run has 8 CUDA kernels, the first copy background image from another device memory location.

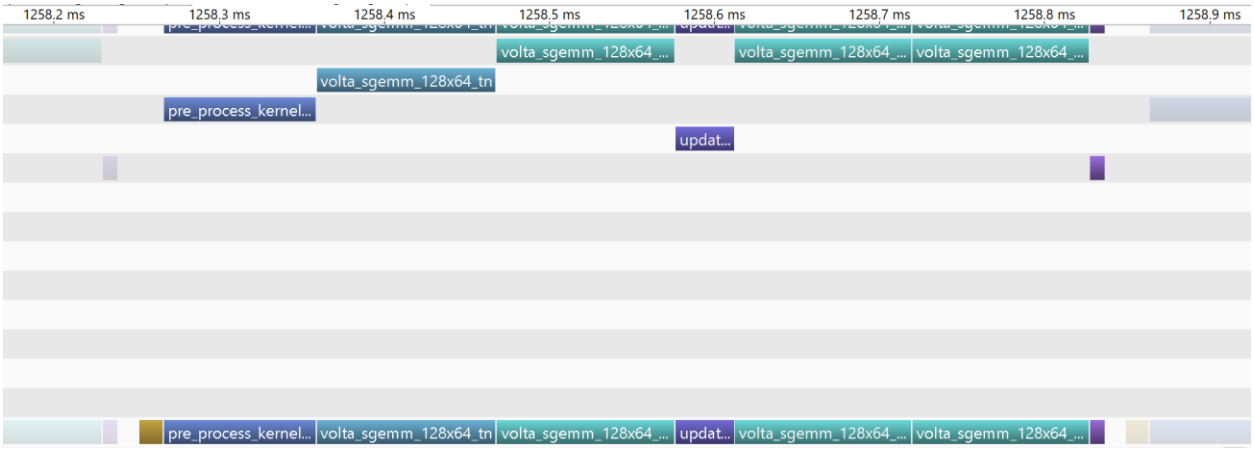


Fig. 8 NVVP performance timeline, after cooperative groups for gradient/Laplacian operator.

2.2. Initialization Optimization

2.2.1. Initialization Optimization

After we have done the optimization steps for computation part in seamless clone feature, from calculating gradient/Laplacian operator, performing Poisson equation solver, to post-processing. The initialization part because the bottle neck, because E2E performance needs to take the initialization into account. After we did a nvprof, it shows in the NVVP timeline, the computation part just consumes very little time, comparing to initialization operations. At the original implementation, its performance is very low.

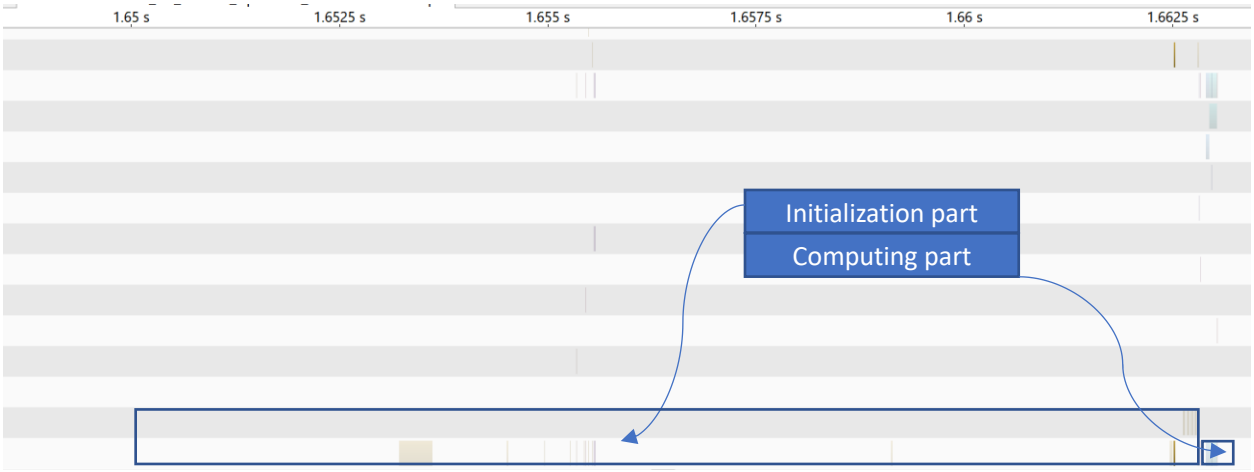


Fig. 9 NVVP performance timeline, before performing optimization for initialization operations.

2.2.2. Optimizing RGBI→RGBP with CPU

The first step in initialization is to convert the data to RGBP format from RGBI format, we originally did this by CPU and copy the converted images to GPU for later processing. We first optimize CPU code and this push down the latency of end2end performance from 29ms to 19ms.

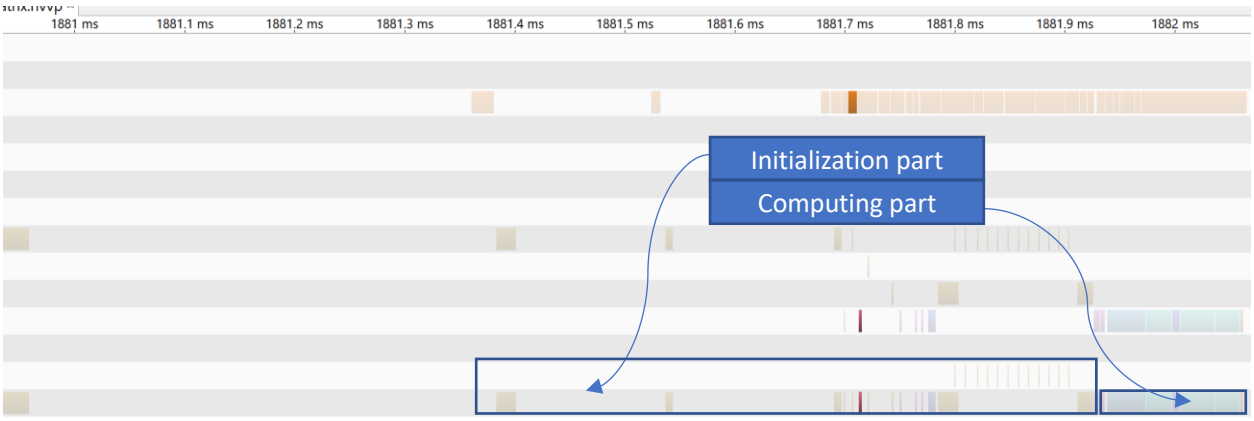


Fig. 10 NVVP performance timeline, after optimizing CPU RGBI→RGBP.

2.2.3. Merged initDSTMatrix

We have another CPU operation which initialize two matrices and two eigen value vectors. We originally did this with CPU and finally move this operation to GPU, with one CUDA kernel. This optimization brings the latency to 14ms.

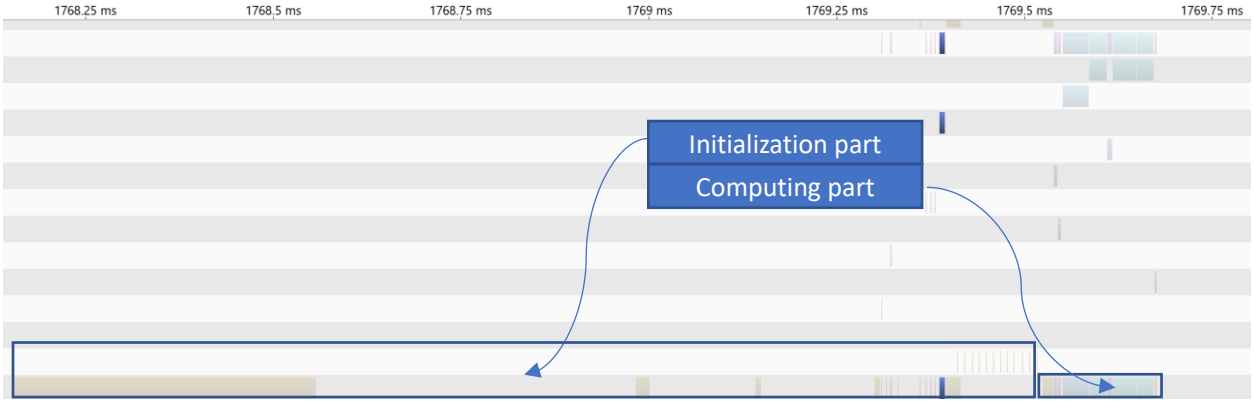


Fig. 11 NVVP performance timeline, after moving matrix/eigen vector initialization to GPU.

2.2.4. Copying Mat with Kernel

We finally replace step 2.2.2, CPU implementation of RGBI→RGBP by GPU, which bring latency down to 0.953ms.

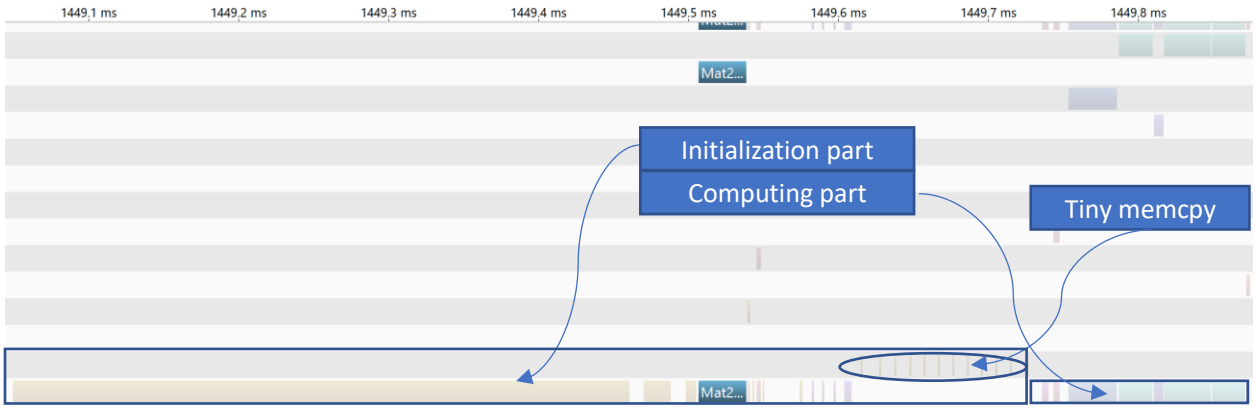


Fig. 12 NVVP performance timeline, after moving RGBI→RGBP to GPU.

2.2.5. Remove Tiny Memcpy

In the last section 2.2.4, you must have noticed that there are several tiny memcpy from host to device, this several memcpy is used by cublasSgemmBatched, we replace this with a CUDA kernel. This brings the latency to 0.768ms.

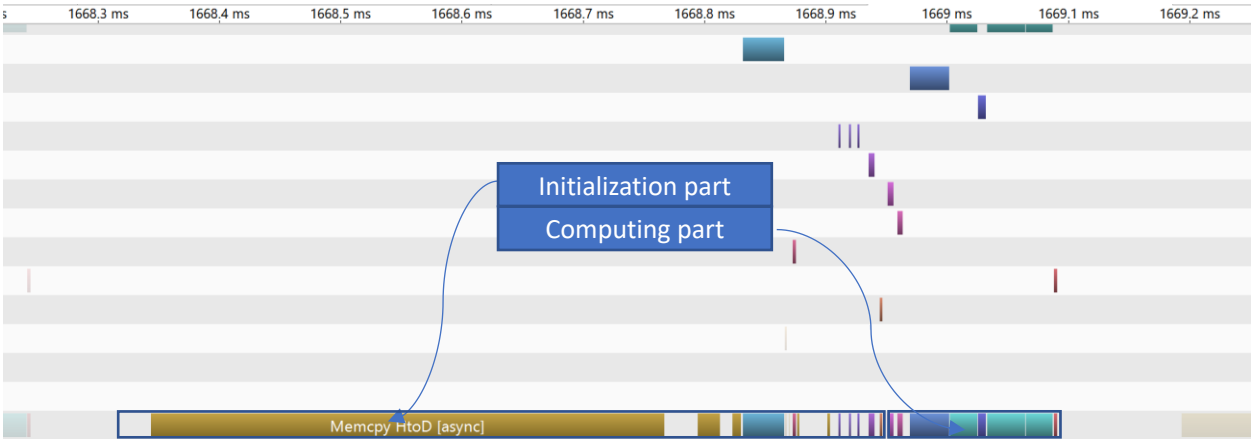


Fig. 13 NVVP performance timeline, after remove tiny memcpy.

2.2.6. Cooperative Groups

We have tried cooperative groups for erode operations, and merge gradient, Laplacian operator into one kernel, but with no performance gain.

After align API inferface with OpenCV	29.422
Optimization conversion RGBI-RGBP with CPU	18.847
Merged initDSTMatrix	14.127
Copying Mat with kernel	0.953

initBlas_resize	0.768
Enable coorperative group	**0.825**

2.3. FFT Accelerated Poisson Equation Solver

Source Patch Size (WidthxHeight)	Performance (ms, 50 rounds with warm up) With/Without FFT		Device Memory (Bytes)	Error(1 in 255) With/Without FFT	
	V100	T4		Diff sum	Diff max
154x100	0.910 / 0.768	2.348 / 1.881	28165184	NA	
300x194	1.270 / 1.270	3.209 / 2.523	34728864	44/128	1/1
592x592	5.423 / 3.736	11.342 / 7.672	79377152	NA	
2400x1552	68.289/ 48.804	149.327 / 81.692	1070380192	17631/ 13659898	1/6