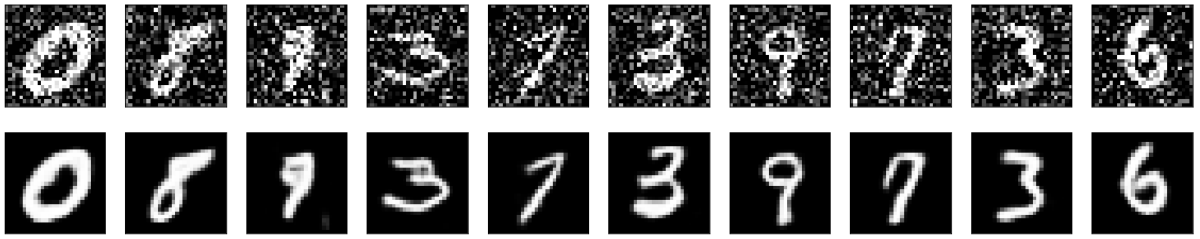
Программирование графических процессоров: Задание #2 (ML)

В качестве задания предлагается решить один из 3х вариантов:

1. Денойзинг изображений из датасета MNIST с помощью свёрточного авто-энкодера.
2. Классификатор изображений из датасета MNIST с помощью свёрточной нейросети.
3. Апроксимация signed distance function геометрии с помощью нейронных сетей и её рендер в реальном времени.

Можно использовать любую технологию программирования GPU.
*Студенты, сдающие курс по вулкану, должны использовать **Vulkan API**.*

Свёрточные нейросети для денойзинга/классификации изображений



Свёрточные нейронные сети часто используются для шумоподавления (денойзинга), а также для сжатия изображения, сохраняя их в пространстве скрытых переменных (latent space). Этот вариант будет интересен тем, кто хотел бы заниматься эффективной обработкой изображений на GPU.

Подготовительная часть

Вариант с денойзингом

1. Изучить архитектуру свёрточных автоэнкодеров и реализовать её на любом удобном фреймворке - pytorch\tensorflow\keras.
2. Загрузить MNIST датасет и сгенерировать зашумленные изображения.
   Именно зашумленные изображения должны будут являться входом наших моделей, а выход нейронной сети должен быть максимально приближен к оригинальным изображениям (смотри изображение выше).
   a. Шумы на изображения необходимо наложить, также как в статье [Keras]: (**проверять работу** вашего алгоритма будут на изображениях мниста, зашумленных этим способом)

```
noise_factor = 0.4
noisy_array = array + noise_factor * np.random.normal(
    loc=0.0, scale=1.0, size=array.shape
)

return np.clip(noisy_array, 0.0, 1.0)
```

3. Обучить модель. Вы можете изучить туториал [Keras] или сделать это любым известным Вам способом.

4. Выгрузить обученную модель любым удобным форматом (onnx). Также допускается выгрузка весов в bin файлы или же иной свой формат.
   Возможно помогут статьи:
   https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html
   https://medium.com/analytics-vidhya/how-to-convert-your-keras-model-to-onnx-8d8b092c4e4f
   Также для экспорта модели из PyTorch для будущего использования в C++ есть удобный способ:
   https://pytorch.org/tutorials/advanced/cpp_export.html#converting-to-torch-script-via-tracing

```
# Encoder
x = layers.Conv2D(32, (3, 3), activation="relu", padding="same")(input)
x = layers.MaxPooling2D((2, 2), padding="same")(x)
x = layers.Conv2D(32, (3, 3), activation="relu", padding="same")(x)
x = layers.MaxPooling2D((2, 2), padding="same")(x)

# Decoder
x = layers.Conv2DTranspose(32, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2DTranspose(32, (3, 3), strides=2, activation="relu", padding="same")(x)
x = layers.Conv2D(1, (3, 3), activation="sigmoid", padding="same")(x)
```

Можно брать архитектуру из данного туториала [Keras]

Вариант с классификацией

По аналогии с вариантом для денойзинга необходимо:

1. Реализовать на выбранном вами фреймворке нейросеть для классификации изображений датасета MNIST.
2. Сохранить веса обученной нейросети в файл.

(подробнее смотри описание варианта для денойзинга)

База

Реализация варианта с денойзингом - **15 баллов**.
Реализация варианта с классификацией - **12 баллов**.

Главным результатом работы должна быть программа denoiser, способная принимать на вход единственный параметр - путь к изображению с .png(.jpg) форматом, которое нужно очистить от шума. (Естественно изображение должно быть из датасета MNIST с наложенным шумом - для тестирования просто выгрузите сгенерированный зашумленный датасет из питона на внешний жесткий диск).

Результатом работы будет являться изображение очищенное от шума и сохраненное по пути исходного с добавлением постфикса перед .png(.jpg) “_denoised”.
Пример:

```
./denoiser “myfolder/im_0_noise.png”
```

после выполнение программы изображение должно появится по пути:

```
myfolder/im_0_noise_denoised.png
```

В варианте с классификацией программа должна выводить в консоль число, изображенное на картинке.

Для чтения и записи .png файлов Вы можете пользоваться любой доступной библиотекой.

Требования к реализации программы:

1. Реализовать вычислительное ядро, позволяющее произвести свёртку ядром размером KxK исходного изображения разрешения WxH и C_in с результирующими C_out каналами. Данный проход должен быть запущен для каждого свёрточного блока нейронной сети.
2. Реализовать вычислительное ядро, накладывающее активационную функцию relu/sigmoid на каждый пиксель каждого канала текущего изображения.
3. Реализовать алгоритм max-pooling на GPU, позволяющий уменьшить разрешение изображения в K раз с поиском максимум-а в блоке размером KxK.

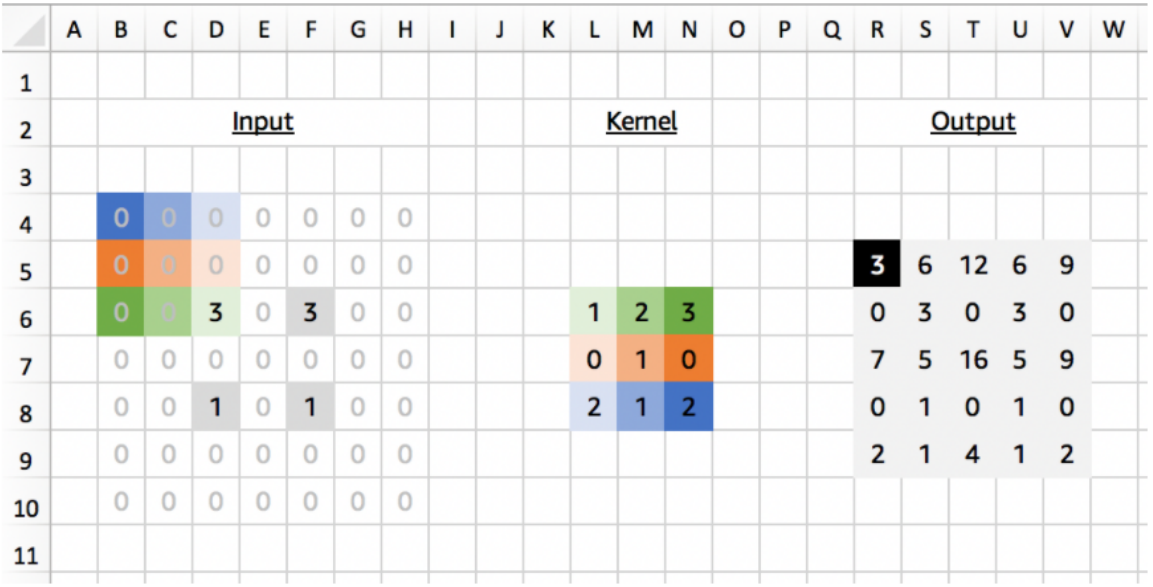


Figure 13: A Conv2DTranspose with 3×3 kernel and stride of 2×2 applied to a 2×2 input to give a 5×5 output.

4. Для decoder-а требуется реализовать обратную свёртку со stride=2.
   Пример обратной свертки со stride=2 (какой и в нашей модели) продемонстрирован в изображение выше. [TransposedConvolution1]
   Обратную свертку можно рассматривать как соединение двух механизмов = конволюции и апсемплинга. Данная комбинация позволяет увеличить исходное изображение с наложением обучаемой интерполяции (конволюции).
   В статье [TransposedConvolution2] приведена интерпретация алгоритма обратной свертки.
5. В итоге у нас должен получиться набор из нескольких пассов:
   a. conv
   b. maxpool
   c. activation(relu\sigmoid)
   d. convtranspose (в случае денойзинга) / linear (в случае классификации)

Используя данный набор пассов можно полностью повторить исходную модель, загружая веса для свёрточных ядер и bias-ы из сохраненной модели. Важно - **можно фиксировать гиперпараметры** - разрешение изображений, stride, размеры kernel-ов - все для облегчения реализации.

В итоге у вас должна получиться программа, умеющая открывать png изображение и исполнять модель на GPU.

Проверка будет происходить следующим образом - проверяющий будет пытаться запустить программу, скомпилированную из исходников **(!!! проверка должна быть тривиальной!!!)** и проводить несколько тестов с собственными случайно выбранными изображениями из датасета MNIST с наложенными шумами. Шумы на изображения будут наложены, также как в статье [Keras]:

```
noise_factor = 0.4
noisy_array = array + noise_factor * np.random.normal(
    loc=0.0, scale=1.0, size=array.shape
)

return np.clip(noisy_array, 0.0, 1.0)
```

Обязательные требования задания

1. Проверить и убедиться в работоспособности программы при работе с изображениями более высокого разрешения - 256x256. Если что-то ломается, то можете переобучить модель на отскалированных изображениях.
2. Добавить режим работы (с помощью аргумента -benchmark X), при котором программа исполнит модель X раз подряд, а также измерит время требуемого для этого. Программа должна вывести итоговое время в ms деленное на число X. Если число слишком маленькое, то можно выводить время требуемое для обработки 10-100 изображений. (У каждого видеокарта своя)

Итоговое решение заливаете в google classroom в виде архива. Можете вести проект на гитхабе а в архиве в readme дать ссылку на Ваш гитхаб.
Напоминаем, внутри архива должны быть:

1. Проект с исходным кодом, который можно тривиально собрать по инструкции.
2. Код для обучения нейросети и сохранения её параметров в файл.
3. Документ с информацией о проведенной работе, оптимизирующих экспериментах и указанием конфигурации Вашей машины.

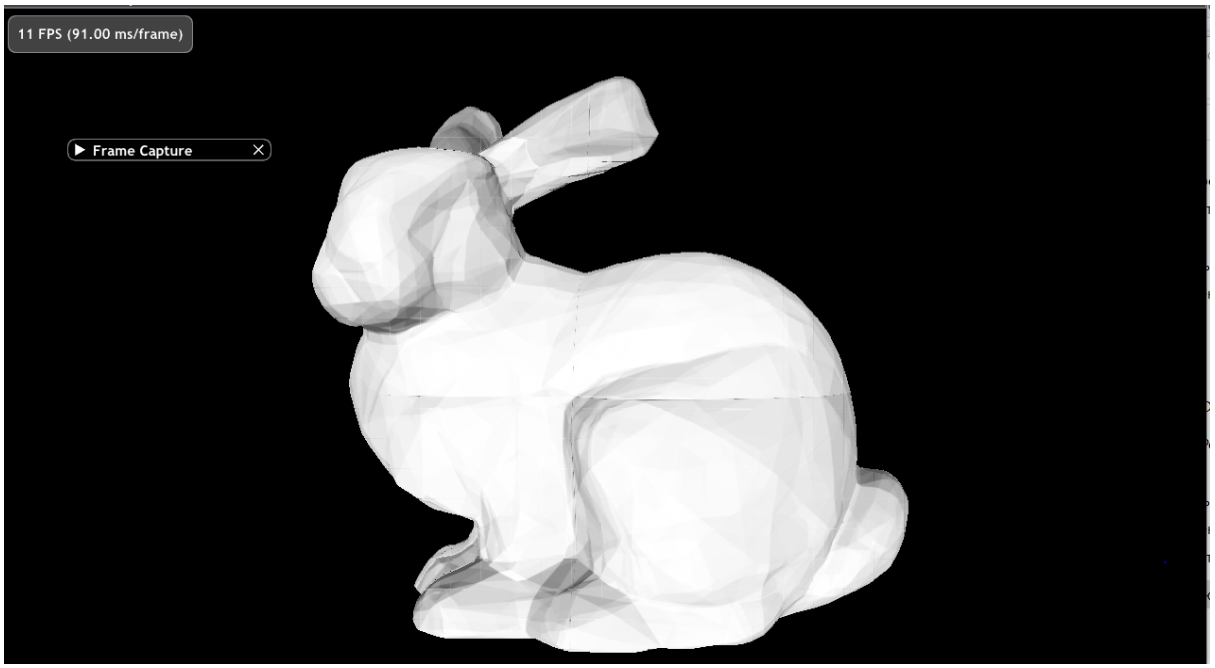
Дополнительная часть

- (+3 Балла) Реализовать **слияние слоёв**: применять активационную функцию не отдельным пассом, а внутри конволюционного пасса для его результата. В чем гипотеза - теперь нашей программе не придется еще один раз читать данные изображения, а также повторно их писать в глобальную память. Приложить к решению документ, в котором будет фигурировать время производительности до и после изменения.
- (+5 Баллов) Реализовать использования local shared memory для кэширования данных изображения и уменьшения числа обращений в глобальную память. Подробнее об этом, а также о том, как можно реализовывать конволюционное ядро для CUDA можно посмотреть в данном примере:

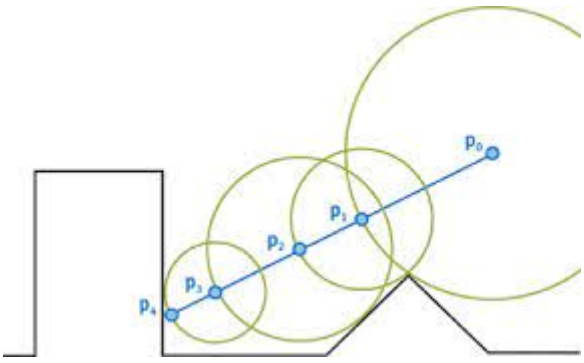
https://github.com/krunal1313/2d-Convolution-CUDA/blob/master/convolution.cu
Приложить к решению документ, в котором будут фигурировать время производительности до и после изменения.

- (+2–5 Баллов) Проведите любую оптимизацию, которую посчитаете полезной и также добавьте в документ информацию о данном своем эксперименты. Гипотезу, мотивацию к ней, а также итоговые экспериментальные данные. Если будет график - вообще супер.
  Советуем посмотреть в сторону распределения потоков для dispatch-а ядра.

Аппроксимация функций расстояния с помощью нейросетей



В этом варианте задания необходимо реализовать алгоритм **Distance Aided Ray Marching** (называемый ещё Sphere Tracing) [RayMarching]. Однако вместо аналитического представления функции расстояний необходимо использовать предобученную нейросеть.



Signed Distance Field (SDF) в точке P с координатами позволяет нам узнать, каково расстояние до ближайшей геометрии. К примеру, если у нас в сцене присутствует в центре сфера радиусом 1, то SDF-ом сцены будет являться аналитическая функция:

SDF(x, y, z) = (x*x+y*y+z*z)^(0.5)-1

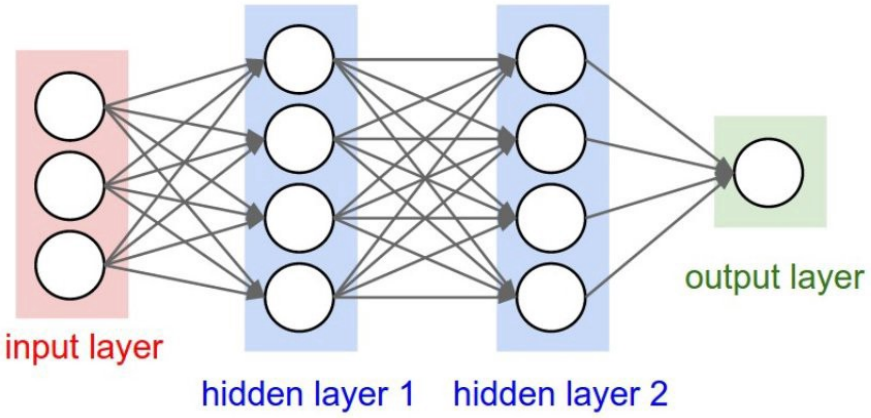
Проблема в том, что аналитически эту функцию сложно вывести для сложных моделей, поэтому в нашем задании мы предлагаем использовать нейронные сети в качестве основы для SDF.

После обучения нейронной сети можно будет её вызывать во всех интересующих точках, для того чтобы узнать расстояние до ближайшей геометрии. Это позволит реализовать алгоритм Sphere Tracing-a, который основан на простой идее:

1. Алгоритм пускает луч из точки P в направление D и хочет узнать итоговую точку пересечения с геометрией.
2. Алгоритм запрашивает значение SDF(P) в точке P и узнает расстояние до ближайшей геометрии. Если расстояние меньше eps (к примеру 0.01, но параметр зависит от сцены, чем он выше - тем выше bias), то мы останавливаем алгоритм. В ином случае алгоритм обновляет значение P = P+D*SDF(P).

Результатом работы алгоритма является последнее значение P. Также следует учесть, что алгоритм может выйти за Bounding Box модели, и чтобы не заниматься бесконечной трассировкой, нужно при выходе за границы останавливать работу алгоритма.
Более подробно про SDF, Sphere Tracing и освещение таких моделей написано в статьях [RayMarching] [SDF].

База



Сложный путь (17 баллов)

1. Реализовать на PyTorch (или ином удобном фреймворке) Multilayer Perceptron с 4 скрытыми слоями и 32 нейронами [PyTorchMLP].
   Активационная функция - LeakyReLU (кроме последнего слоя). На вход оно должно принимать 3 параметра - позицию - x,y,z. На выходе предсказывать расстояние до ближайшей поверхности модели. Это расстояние должно может быть положительно, если точка находится вне модели, и отрицательно если внутри модели. Если точка находится на поверхности модели, то расстояние равно 0. Более подробно можно прочесть в статье - https://arxiv.org/pdf/2009.09808.pdf

2. Сгенерировать данные SDF для экспериментальной модели. В качестве модели желательно использовать Stanford Bunny [Bunny]. Также рекомендуем использовать библиотеку mesh_to_sdf и её функцию sample_sdf_near_surface для генерации датасета (прошу заметить, что её зависимость - pyrender - лучше установить версией 0.1.39: pip install pyrender==0.1.39, в ином случаи могут быть проблемы с исполнением некоторых функций на Windows).
3. Выгрузить обученную модель любым удобным форматом (onnx). Также допускается выгрузка весов в bin файлы или же иной свой формат, хоть текстовый.

Возможно помогут статьи:
https://pytorch.org/tutorials/advanced/super_resolution_with_onnxruntime.html
https://medium.com/analytics-vidhya/how-to-convert-your-keras-model-to-onnx-8d8b092c4e4f
Также для экспорта модели из PyTorch для будущего использования в C++ есть удобный способ:
https://pytorch.org/tutorials/advanced/cpp_export.html#converting-to-torch-script-via-tracing

Легкий путь (14 баллов):

Уже есть реализованная и обученная модель (с зайчиком) с такой архитектурой:

```
nn.Linear(3, 32),
nn.LeakyReLU(0.1),
nn.Linear(32, 32),
nn.LeakyReLU(0.1),
nn.Linear(32, 32),
nn.LeakyReLU(0.1),
nn.Linear(32, 32),
nn.LeakyReLU(0.1),
nn.Linear(32, 1),
nn.Tanh()
```

На вход подается линейный вектор с координатами x, y, z, на выходе после tanh нейронка выдает результирующий sdf.
Есть обученные веса модели, разложенные по файлам:
https://drive.google.com/drive/folders/1PsdSXVVqZf8F-hXFlB1qPeob7iVijsjG?usp=sharing
weights<x>.txt и biases<x>.txt - содержат веса для x-го слоя сети. Веса разложены в линию. К, примеру, для того чтобы посчитать значения 0-го нейрона 0-го скрытого слоя, потребуется вычислить выражение: leakyRelu(x*0.1028+y*1.6673+z*1.4198-0.3867) (откройте файлы weights0.txt+biases0.txt для понимания).
Вам остается лишь прочесть эти веса и загрузить в память видеокарты, тем или иным образом.

Исполнение нейронной сети

1. Реализовать чтение модели в C++ часть.
2. Загрузить сначала в CPU память, а далее в GPU буфер веса нейронной сети.
3. Реализовать экранный алгоритм sphere-tracing-а на основе компьют шейдера, где функция SDF аппроксимируется нейронной сетью. Для этого требуется реализовать матричные перемножения нейронов с помощью двойных вложенных циклов, а также активационную функцию LeakyReLU. Результатом работы алгоритма должна являться визуализация глубины.
   ВАЖНО: параллельность достигается просто на уровне пикселей. Другими словами, каждый “поток” GPU должен исполнять нейронную сеть самостоятельно. Не требуется пытаться ускорить исполнение нейронной сети для 1 запроса несколькими потоками GPU.

Итоговое решение заливаете в google classroom в виде архива. Можете вести проект на гитхабе а в архиве в readme дать ссылку на Ваш гитхаб.
Напоминаем, внутри архива должны быть:

1. Проект с исходным кодом, который можно тривиально собрать по инструкции.
2. Код для обучения нейросети и сохранения её параметров в файл (если вы пошли более сложным путём).
3. Документ с информацией о проведенной работе, оптимизирующих экспериментах и указанием конфигурации Вашей машины.

Дополнительная часть

- (+2 балла) Реализовать вычисление нормалей видимой поверхности для каждого пикселя, благодаря явному расчету производных и использованию local shared memory, внутри этого же самого шейдера со sphere tracing-ом. Визуализировать нормали.
  Не забудьте обработать качественно края warp group, чтобы и у них были адекватно рассчитанные нормали.
- (+2 балла) Добавить освещение от направленного источника света - ламбертовый шейдинг (косинус - нужна нормаль + учесть тень).
  1. Для реализации данного пункта вам нужно лишь определить функцию isInShadow(P, D), которая будет возвращать значение описывающее сколько доходит света до точки P по направлению (падающего света) D.
  2. Также учтите, что может возникнуть нежелательный эффект затенения собственной поверхностью из-за float арифметики. Попытайтесь разрешить данную проблему.
- (+2-3 балла) Реализовать кэширование весов нейронной сети в local shared memory, и использование их для трассировки вместо обращения к глобальной памяти. Обратите внимание что в этом варианте Вам скорее всего придётся использовать несколько потоков на 1 луч. Сравнить производительность. Объяснить результаты (они могут различаться на разных видеокартах, в связи с разной архитектурой кэш систем).
- (+4 балла) Визуализация изображения в реальном времени в окне: необходимо использовать графические API или interop. При копировании данных в хостовую память доп. баллы засчитаны не будут!
- (+1 балл) управление камерой (перемещение и вращение) / возможность вращать направление солнца.

Литература

- [Keras] Santiago L. Valdarrama. Convolutional autoencoder for image denoising. https://keras.io/examples/vision/autoencoder/
- [PyTorchMLP] https://medium.com/biaslyai/pytorch-introduction-to-neural-network-feedforward-neural-network-model-e7231cff47cb
- [TransposedConvolution] Thom Lane. https://medium.com/apache-mxnet/transposed-convolutions-explained-with-ms-excel-52d13030c7e8 Transposed Convolutions explained with… MS Excel!
- [TransposedConvolution2] https://towardsdatascience.com/understand-transposed-convolutions-and-build-your-own-transposed-convolution-layer-from-scratch-4f5d97b2967
- [RayMarching] Фролов В.А. Трассировка лучей с использованием функций расстояний http://ray-tracing.ru/articles231.html
- [SDF] http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/

- [Bunny] https://graphics.stanford.edu/~mdfisher/Data/Meshes/bunny.obj