ChatGPT

Архитектура NVIDIA Jetson Orin NX 16GB: память и организация пайплайна DeepStream

Аппаратная архитектура Jetson Orin NX 16GB

Jetson Orin NX 16GB – это мощный одноплатный модуль (SoC) с интегрированными CPU, GPU и общей памятью. Ключевые аппаратные компоненты Orin NX 16GB следующие:

- **GPU**: Графический процессор архитектуры NVIDIA Ampere с 1024 CUDA-ядрами и 32 тензорными ядрами [1], работающий на частоте до ~918 МГц. Теоретическая производительность AI достигает ~100 TOPS (INT8,_sparse) [2]. GPU поддерживает современные графические API (OpenGL 4.6/Vulkan 1.1) и имеет улучшенное сжатие данных и кеширование для высокой пропускной способности [3].
- **CPU**: Восемь ядер ARM Cortex-A78AE (64-бит, ARMv8.2) с тактовой частотой до ~2 ГГц [4]. Ядра сгруппированы в кластеры с общим кэш-памятью L3 4 МБ [4]. Этот процессор отвечает за общесистемные задачи и координацию работы GPU.
- **Память**: 16 ГБ LPDDR5 SDRAM с 128-битной шиной [5]. Память высокоскоростная (частота до 3200 МГц) с пиковой пропускной способностью ~102 ГБ/с [6]. Вся память является **унифицированной**, т.е. **общей для CPU и GPU** – у модуля нет отдельной дискретной VRAM, вместо этого и CPU, и GPU обращаются к одному общему пулу DRAM.

Такое сочетание обеспечивает компактность и энергоэффективность: все компоненты расположены на одном кристалле (SoC) или модуле, что минимизирует задержки передачи данных. Помимо CPU и GPU, Orin NX включает два аппаратных ускорителя глубинного обучения NVDLA (каждый по ~20 TOPS в INT8) [7] для выполнения инференса с минимальной нагрузкой на GPU, а также аппаратные кодеки видео (NVENC/NVDEC) для кодирования/декодирования H.264, H.265, AV1 и др. [8].

Практическое значение архитектуры: Орин NX предназначен для задач AI на краю (Edge AI): робототехника, видеонаблюдение, автономные системы. Его аппаратные ресурсы позволяют одновременно декодировать несколько потоков видео, выполнять нейросетевой анализ (компьютерное зрение) на GPU/DLA и выводить результаты на дисплей – всё на одном устройстве. Ниже мы рассмотрим, как именно архитектура памяти Jetson Orin NX влияет на организацию данных между CPU и GPU и на построение высокопроизводительного пайплайна обработки видео с NVIDIA DeepStream.

Единая память CPU и GPU: как организована и как разделяется

Jetson Orin NX (как и другие SoC NVIDIA Tegra) использует **унифицированную память** – это означает, что **и CPU, и GPU разделяют одну физическую память DRAM** без разделения на системную и видеопамять [9]. В отличие от классического ПК с дискретной видеокартой (где у GPU своя VRAM, соединённая через PCIe/NVLink), здесь общий 16-гигабайтный пул доступен обеим подсистемам. **Нет жёстко выделенной доли памяти под GPU или CPU** – операционная система и драйверы

динамически распределяют память по запросам. Поэтому, если CPU занял значительный объём ОЗУ, этот же объём **уже недоступен GPU** для размещения текстур, тензоров и пр. [10]. Фактически, **GPU-видимая память = Свободная системная память**, и разработчику нужно учитывать общий расход.

Важно понимать, что **общая память – не означает отсутствие контроллеров и кешей**. У CPU и у GPU есть отдельные контроллеры памяти (Memory Controller) для доступа к DRAM [11]. Кроме того, у каждого свои кеш-иерархии (у CPU – кеш L1/L2/L3, у GPU – собственные L1/L2 для текстур/данных). **Согласованность (coherency)** между CPU и GPU кешами поддерживается аппаратно начиная с поколения NVIDIA Xavier/Orin – GPU способен видеть обновления данных, сделанные CPU, даже если они ещё в кешах CPU (т.н. **I/O coherency**, реализовано для GPU с compute capability $\geq$7.2) [12] [13]. Более того, Orin поддерживает и обратную когерентность (CPU может читать обновления из кеша GPU), что снижает накладные расходы на ручное сброс кешей при обмене данными [14]. Благодаря этой архитектуре, при правильном подходе можно напрямую **делить буферы** между CPU и GPU без копирования – например, GPU может обработать фрейм, который декодирован CPU, и наоборот, **в пределах одной общей адресной зоны**.

Однако **распределение адресного пространства** между CPU и GPU всё же логически разделено. GPU работает со своими виртуальными адресами через GMMU/SMMU (системный MMU) и может адресовать любую физическую страницу DRAM, но только если она выделена как доступная GPU (например, через cudaMalloc или через специальные API драйвера). В ОС по умолчанию часть адресов может быть зарезервирована под DMA-буферы, драйверы (например, для камеры или дисплея). Но в целом разработчик не задаёт вручную «сколько памяти отдать GPU» – используется единый пул, и **эффективное разделение** достигается за счёт правильного управления буферами.

Вывод: Память Jetson Orin NX **общая и гибко используемая**: это упрощает обмен данными (нет узкого места PCIe), но требует внимания к конкурентному потреблению ресурсов CPU/GPU и к синхронизации доступа, о чём далее.

Почему копирование данных между CPU и GPU дорого даже при общей памяти

Может показаться, что раз память единая, то *копировать* данные между CPU и GPU вовсе не нужно – достаточно предоставить указатель на один и тот же буфер. В идеале, так и есть: можно, например, выделить “Zero Copy” буфер (page-locked host memory или cudaHostAllocMapped) и GPU сможет читать/писать напрямую в него [10]. **DeepStream** и GPU-ускоренные фреймворки обычно используют буферы в таком “визируемом” GPU-формате (тип памяти NVMM, см. ниже), чтобы избежать лишних копий. Тем не менее, на практике *полностью избежать перемещения данных* сложно, и вот почему:

- **Объём данных и пропускная способность памяти.** Любое копирование – это чтение и запись большого блока DRAM. Хотя в Orin NX память быстрая (~102 ГБ/с), 4K/UHD кадры или тензоры могут занимать сотни мегабайт, и перемещение их даже по внутренней шине занимает миллисекунды. При частых копиях это время складывается и отнимает ценное время GPU. Одновременный доступ CPU и GPU делят ту же шину DRAM, **конкурируя за полосу** (I/O bandwidth) [15]. Поэтому избыточные копии “проедают” память и снижают общую производительность.

- **Синхронизация кешей и доступов.** Несмотря на аппаратную когерентность, драйверу всё равно приходится координировать доступ. Если мы используем **Unified Memory (кумулятивную память)** CUDA, драйвер **под капотом создаёт две копии** данных – в адресном пространстве CPU и GPU – и синхронизирует их при обращениях [16]. Это упрощает жизнь программисту (CUDA сама мигрирует страницы между CPU↔GPU по запросу), но может вести к непредвиденным затратам: например, страница может быть скопирована из GPU обратно в CPU RAM при обращении CPU. **NVIDIA отмечает**, что unified memory **может работать медленнее**, так как требует поддержания копий и фоновой синхронизации между CPU и GPU [17]. И хотя вам не нужно вручную заботиться о согласованности, **драйвер тратит время на обеспечение этой согласованности**, иногда блокируя выполнение ядра или занимая канал памяти [16].
- **Необходимость преобразований.** В реальных приложениях данные могут требовать преобразования форматов (например, BGR → RGB → YUV), изменения размещения (из разбросанных буферов камеры в сплошной массив) или выравнивания. Часто эти задачи выполняются на CPU с созданием нового буфера, который потом передаётся GPU – а это копирование. Даже на самом Jetson, если мы некорректно построим конвейер (например, заставим GPU-буфер «вернуться» в системную память), случится дорогостоящий цикл: GPU должен сбросить данные в DRAM, CPU их прочитать и снова отправить на GPU. Поэтому правильная обработка **“на месте” (in-place)** или **с нулевыми копиями (zero-copy)** – ключ к производительности.

Вывод: Операции передачи данных между CPU и GPU на Jetson дорогостоящи в первую очередь из-за **ограниченной полосы памяти** и **накладных расходов на когерентность данных**. Несмотря на отсутствие интерфейса PCIe, пропускная способность памяти не безгранична, и любые ненужные перемещения данных снижают доступный bandwidth для вычислений. Чтобы полностью раскрыть потенциал Orin NX, надо максимально избегать копирований – особенно больших кадров – и хранить/обрабатывать их непосредственно на GPU, используя оптимальные механизмы (память типа NVMM, pinned memory, unified memory с учётом особенностей и т.д.).

Эффективная работа с памятью в DeepStream-пайплайне

Архитектура Jetson Orin NX отлично подходит для работы NVIDIA DeepStream SDK – фреймворка для ускоренной обработки видео с помощью GStreamer-пайплайнов. **DeepStream 7.1 (JetPack 6.2)** уже поддерживает Orin NX и включает набор GStreamer-плагинов (аппаратный декодер/кодер, композитор, инференс, OSD и др.), оптимизированных под NVIDIA GPU. Рассмотрим, **как организовать конвейер (pipeline)** с двумя ветвями – **анализ (инференс)** и **отображение (стриминг)** – так, чтобы обе ветки могли *разделять одни и те же кадры в GPU-памяти без лишних копирований*.

Сценарий: Имеются два видеоисточника (например, две камеры MIPI CSI или видеофайлы «левая» и «правая»), которые необходимо **сшить в панораму**. На полученном широком кадре (размером 6528×1632) выполняется инференс нейросети (например, детекция объектов моделью YOLOv11n/s в TensorRT). Одновременно тот же видеопоток нужно **вывести на экран/стрим с задержкой ~7 секунд** (возможно, для буферизации или наложения результатов анализа). Важно сделать это эффективно: не дублировать кадры в памяти CPU, а хранить их единым буфером, доступным и для анализа, и для вывода.

Разделение конвейера на две ветки (analysis/display)

В GStreamer для разветвления потока используется элемент `tee`, который позволяет одному входному буферу иметь несколько потребителей. DeepStream-плагины поддерживают работу с **управляемыми NVIDIA буферами (NVMM)**, поэтому при правильном построении *кадры могут разделяться между ветками без копирования*. Как отмечает NVIDIA, при использовании `tee` **видеокадры шарятся между разными выходными ветвями** – фактически обе ветки получают указатель на один и тот же NVMM-буфер [18]. Это возможно, пока ни одна из веток не пытается изменить содержимое буфера или принудительно скопировать его в другую память.

Чтобы ветки работали независимо, обычно сразу после `tee` ставят `queue` в каждую ветку (разделяя потоки выполнения). При этом крайне важно, **где** размещён queue: если поместить его после элемента, который возвращает данные в CPU, то буфер в очереди уже будет системной памятью (что нежелательно). Нужно стараться, чтобы все элементы до самого вывода в обеих ветках поддерживали `memory:NVMM` – тогда данные останутся на GPU.

Типичный пайплайн для нашего сценария может выглядеть так:

1. **Декодирование и сшивание:** оба источника поступают на аппаратный декодер (`nvv4l2decoder`), отдающий кадры в NVMM. Затем с помощью GPU-композитора (например, `nvcompositor` или `nvmultistreamtiler`) два изображения объединяются в одно панорамное [19]. На выходе композиции – широкий кадр в NVMM-памяти.
2. **Ветвление потоков:** панорамный кадр идёт в `tee`, который разделяет поток на две ветки. **Ветка A (анализ)**: кадр идёт в инференс. **Ветка B (отображение)**: кадр идёт в буфер задержки и затем на вывод.
3. **Ветка инференса:** сначала кадр может пройти через `nvstreammux` (обязательно, если инстансы инференса работают батчем) – он формирует пакет, даже для одного потока. Далее, **разбиение на тайлы 1024×1024**. Это можно сделать двумя способами:
4. **Через плагин предобработки**: DeepStream имеет модуль `gst-nvdspreprocess`, позволяющий указать несколько областей (ROI) на одном кадре и автоматически нарезать их под вход нейросети [20]. Например, в конфигурации можно задать 6 ROI 1024×1024, и плагин подготовит шесть обрезанных подпотоков (тензоров) для `nvinfer` – всё это происходит на GPU без участия CPU [21].
5. **Через несколько инстансов nvinfer:** альтернативно, можно запустить несколько `nvinfer` (с одинаковой моделью YOLOv11) и каждому подать кадр с разным параметром ROI-кропа. Однако это сложнее координировать. Предпочтительно использовать встроенный `nvdspreprocess` (начиная с DeepStream 6.0+) для нарезки по тайлам [22].

Модель YOLO (например, Yolov11n/s в TensorRT) принимает на вход эти 1024×1024 изображения и выдаёт детекции. Результаты привязываются к оригинальному кадру через метаданные (NvDsFrameMeta/NvDsObjectMeta). 4. **Ветка отображения с задержкой:** выход `tee` для ветки B идёт через небольшой `nvvideoconvert` (если нужно, например, сконвертировать NV12→RGBA на GPU для отображения). **Важно:** использовать `nvvideoconvert` (аппаратный) вместо стандартного CPU-`videoconvert`, иначе буфер выйдет из NVMM и копирование неизбежно [23]. После конвертации ставится `queue` с настроенными параметрами буферизации: например, `max-size-buffers` или `max-size-time` рассчитаны на ~7 секунд видео (при 30 FPS это порядка 210 кадров). Эта очередь и создаёт задержку: она копит кадры в GPU-памяти и отдаёт вниз по течению с нужным

запаздыванием. NVIDIA рекомендует помещать `queue` **сразу после nvvideoconvert**, пока данные ещё в NVMM, чтобы очередь хранила именно GPU-буферы [24]. В нашем примере `queue` будет содержать до ~210 NVMM-кадров, что потребует значительного количества памяти и GPU-буферов. 5. **Объединение результатов и вывод:** спустя 7 секунд задержки каждый кадр из очереди ветки B достигает отображения. Перед выводом можно наложить результаты инференса: например, плагин OSD (`nvdsosd`) возьмёт метаданные детекций, прикреплённые к кадру, и нарисует рамки/лейблы. Поскольку инференс происходил на исходном кадре, метаданные нужно правильно пробросить – DeepStream делает это через общий GstBuffer и умные метаданные. **Если же ветки полностью раздельны**, можно вместо OSD воспользоваться собственным обработчиком: например, по идентификатору кадра найти соответствующие детекции и отрисовать их. Наконец, кадр выводится либо на дисплей (`nveglglessink` поддерживает входные NVMM-буферы), либо отправляется в сетевой поток (через энкодер `nvv4l2h264enc` → `rtspclientsink` и т.д.).

Буферизация на GPU и управление памятью

Организация 7-секундной задержки на GPU накладывает требования на **память и буферные настройки**. Как мы выяснили, `queue` не выполняет никаких преобразований — он просто хранит входящие буферы. Если на вход очереди поступают NVMM-буферы, то и хранятся они **в виде NVMM (CUDA)** памяти. Форум NVIDIA подтверждает, что **NVMM-кадры могут буферизоваться** в очереди, если не вставлять преобразование в системную память [24]. В приведённом выше пайплайне мы позаботились об этом, используя nvvideoconvert вместо videoconvert. Разработчики DeepStream указывают, что по умолчанию аппаратный декодер оперирует пулом из 16 кадров (buffer-pool), которые переиспользуются [25]. Однако при 7-секундной задержке 16 буферов явно недостаточно – очередь будет накапливать до сотен кадров. Поэтому может потребоваться увеличить размер пула декодера (параметр `num-extra-surfaces` у nvv4l2decoder) и самого queue (`max-size-buffers`), иначе pipeline может остановиться в ожидании свободного буфера. Будьте готовы и к росту потребления памяти: хранить десятки или сотни необжатых кадров 6528×1632 (NV12) – это гигабайты данных (1 кадр ≈ 10 Мпикс * 1.5 байт ≈ 15 МБ; 200 кадров ≈ 3 ГБ). **Аппаратная память Orin NX должна быть достаточна (16 ГБ), но нужно учитывать и другие нагрузки (TensorRT буферы, модель, ОС)**.

Хорошей новостью является то, что такая буферизация почти не трогает CPU. **Все операции происходят на GPU**: декодер кладёт кадры в GPU-память, `tee` раздаёт указатели, queue держит их там, а вывод может брать кадры напрямую для отображения. Таким образом, мы **избегаем лишних копирований между CPU/GPU**, достигая цель – *общий буфер кадров для обеих ветвей*. В официальном обсуждении DeepStream подчеркнуто, что убрав лишние преобразования, можно реализовать задержку **полностью внутри NVMM-бферов** и тем самым не платить ценой копий [26] [23].

Рекомендации для реализации пайплайна

- **Используйте NVIDIA-плагины для видеоопераций.** В цепочке конвейера старайтесь применять `nvv4l2decoder`, `nvvideoconvert`, `nvstreammux`, `nvinfer`, `nvdsosd` и пр. – эти элементы работают с памятью NVMM и поддерживают DMAbuf обмен без копий. Избегайте стандартных GStreamer-элементов, которые не поддерживают память:NVMM (например, `videoconvert`, `appsink` без специальных настроек) – они вынудят перенос в системную память.

- **Передавайте данные между ветками через** `tee` **+** `queue`**.** Этот паттерн обеспечит независимую обработку в параллельных ветвях, не дублируя сами данные. Как упоминалось, `tee` не копирует буфер, а увеличивает его ссылочный счетчик, позволяя раздать несколько ссылок [18]. Благодаря этому инференс-ветка и отображение-ветка увидят один и тот же кадр.
- **Увеличьте количество буферов под задержку.** Настройте decoder (`num-extra-surfaces`) и `queue` (`max-size-buffers`, `max-size-bytes`, `max-size-time`) так, чтобы они вместили 7 сек видео. Важно также выставить `queue leaky=0` (по умолчанию) чтобы кадры не терялись, либо `leaky=2` (latest) если допустимо отбрасывать старые кадры при переполнении.
- **Синхронизация потоков и метаданных.** Если отображение задержано, а анализ идёт без задержки, придётся аккуратно совмещать результаты. Один из подходов – хранить результаты инференса и привязывать их к идентификатору/номеру кадра. DeepStream использует `frame_id` и метаданные, так что можно, например, при выводе кадра (через 7 сек) обратиться к сохранённым результатам анализа этого кадра и наложить их. Альтернативно, можно *провести обе ветки через общий узел OSD перед выводом*: для этого вместо полностью раздельных веток их можно снова объединить (например, через `nvstreammux` / `nvstreamdemux` после задержки) – однако это усложняет пайплайн. В большинстве случаев проще связать метаданные по `frame_number`. Сама же задержка не повлияет на качество анализа – модель обработает кадры своевременно, просто их отображение отложено.

Заключение

NVIDIA Jetson Orin NX 16GB предоставляет высокую производительность благодаря интеграции CPU, GPU и памяти на одном модуле. Его **общая память** позволяет эффективно обмениваться данными между CPU и GPU, но разработчик должен учитывать ограничения по пропускной способности и избегать ненужных копирований. В реализациях на **DeepStream** это достигается использованием аппаратно-ускоренных плагинов и продуманной архитектуры конвейера: хранение видеофреймов на GPU, параллельное ветвление потоков и буферизация задержки реализуются без участия CPU. Рассмотренный пример (стерео-панорама с инференсом на тайлах и 7-секундным отложенным выводом) демонстрирует, что при правильной организации можно **обрабатывать поток в реальном времени** на Orin NX, максимально используя его потенциал. Такой подход минимизирует задержки и разгрузку памяти, позволяя системе решать сложные задачи компьютерного зрения на краевом устройстве.

Sources:

1. NVIDIA Developer Forums – Memory architecture on Orin (unified memory for CPU/GPU) [9] [10]
2. NVIDIA Jetson Orin NX Datasheet – Hardware specifications (GPU, CPU, memory bandwidth) [1] [5] [6]
3. NVIDIA Developer Forums – DeepStream pipeline branching and NVMM buffering [18] [24]
4. NVIDIA Developer Forums – Unified Memory on Jetson (performance considerations) [17] [16]
5. NVIDIA DeepStream Documentation – Preprocessing plugin (ROI-based tiling for inference) [20]

---

[1] [2] [3] [4] [5] [6] [7] [8] connecttech.com
https://connecttech.com/pdf/jetson_orin_nx_datasheet.pdf

9 12 13 14 1. CUDA for Tegra — CUDA for Tegra 13.0 documentation
https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html

10 Memory Architecture Differences in x86 and SoC GPUs - Jetson Orin Nano - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/memory-architecture-differences-in-x86-and-soc-gpus/258135

11 Jetson Orin NX: Shared memory between CPU and GPU - Jetson Orin NX - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/jetson-orin-nx-shared-memory-between-cpu-and-gpu/344025

15 16 17 Like cudaMemcpy is slow on orin nx - Jetson Orin NX - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/like-cudamemcpy-is-slow-on-orin-nx/262079

18 19 23 24 25 26 Put delay after streammux - DeepStream SDK - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/put-delay-after-streammux/315966

20 21 Gst-nvdspreprocess — DeepStream documentation
https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_plugin_gst-nvdspreprocess.html

22 Pre-processing config in deepstream 6.0 - DeepStream SDK - NVIDIA Developer Forums
https://forums.developer.nvidia.com/t/pre-processing-config-in-deepstream-6-0/220316