NVIDIA DGX Spark Porting Guide

NVIDIA Corporation

Feb 17, 2026

Overview

To reach new levels of productivity and AI performance, the NVIDIA DGX Spark platform combines high-performance, efficient ARM cores with a powerful integrated Blackwell GPU with unified memory.

This guide is also available for download as a PDF.

Chapter 1. System Overview

NVIDIA DGX Spark offers a robust and versatile hardware and software environment tailored for advanced AI applications.

1.1. Hardware

This section provides an overview of the key hardware differences between systems based on x86_64 CPUs with discrete GPUs (dGPUs) and our ARM-based System-on-Chip (SoC) platform.

1.1.1. CPU Architecture

In comparison to the classical x86_64 processor in combination with a dedicated GPU, our ARM SoC combines CPU, GPU, and other accelerators on one chip. The CPU is a 20-core ARM64-based chip, sharing 128GB of LPDDR5x memory with the integrated GPU (iGPU). Our ARM SoC features a hybrid architecture consisting of 2 clusters.

Each cluster has 5 high-performance cores (ARM v9.2, Cortex-X925) with 2MB L2 cache each, and 5 high-efficiency cores (ARM v9.2, Cortex-A725) with 512KB L2 cache each. The high-performance cluster features 16MB L3 cache, while the high-efficiency cluster is equipped with an 8MB L3 cache.

This configuration allows for dynamic workload management, optimizing power consumption and thermal performance, and enables workload-specific optimizations that balance performance and efficiency.

Memory Model Differences

Another critical difference lies in the memory models. The x86_64 architecture is known for its strict memory consistency, ensuring that memory operations are highly predictable and synchronized across cores. This rigidity can simplify programming but may introduce performance bottlenecks due to the need for frequent synchronization. Conversely, ARM architectures offer a more relaxed memory model, which can lead to synchronization challenges if not managed properly. This flexibility allows for performance optimizations, particularly in scenarios where memory operations can be reordered for efficiency.

For more details see *ARM Memory Ordering*.

1.1.2. GPU

The GPU integration differentiates our ARM-based System-on-Chip (SoC) platform from traditional x86_64 systems with discrete GPUs.

Our ARM SoC integrates a NVIDIA Blackwell Architecture GPU, which shares the 128GB of LPDDR5x memory with the CPU. Unlike other vendors, which use a fixed carve-out for the iGPU in the shared memory, we utilize a dynamic unified memory architecture (UMA), allowing both the CPU and iGPU to access the same memory space. This integration enables efficient data sharing by eliminating the need for data copies between CPU RAM and VRAM. Integrated GPUs, such as those in the DGX Spark, are optimized for lower power consumption, making them ideal for mobile and embedded applications.

GPU Specification

The integrated Blackwell GPU on our ARM SoC includes 5th generation Tensor Cores, 4th generation RT Cores, 1x NVENC and 1x NVDEC.

1.1.3. System Architecture (Memory and Buses)

Another key difference between x86_64 + dGPU systems and our ARM SoC is the memory and bus architecture.

Memory Hierarchy

In traditional x86_64 systems, memory is distinctly separated between the CPU and the GPU. The CPU accesses system RAM, while the GPU utilizes dedicated video RAM (VRAM). This separation allows each component to optimize its memory usage for specific tasks, but it also introduces the overhead of transferring data between CPU and GPU memory.

ARM-based SoCs, including the DGX Spark platform, employ a Unified Memory Architecture (UMA). In UMA, both CPU and GPU share the same physical memory space without a fixed carve-out. This design allows for data sharing between CPU and iGPU, reducing latency and eliminating the need for redundant data copies. The shared memory model enhances performance in workloads that require frequent CPU-GPU collaboration, as data can be accessed seamlessly by both processing units.

Bus Architecture

ARM SoCs like the DGX Spark benefit from an integrated bus architecture with a 256-bit memory interface and 273GB/s bandwidth, facilitating efficient communication between CPU, GPU, and other components. This contrasts with the often more complex and power-intensive bus systems in x86_64 architectures, which must manage separate CPU and GPU components.

1.1.4. Peripherals, Networking, Connectivity

The DGX Spark platform supports a range of peripherals and networking options. It includes 1x RJ-45 connector with 10 GbE Ethernet, a ConnectX-7 Smart NIC, WiFi 7, and Bluetooth 5.3 with LE. Additionally, it features 4x USB Type-C ports and 1x HDMI 2.1a display connector.

1.2. Software

DGX OS is based on Ubuntu 24.04 (LTS) with added NVIDIA drivers, libraries, frameworks and tools.

For more details see *DGX Spark Software Stack*.

Chapter 2. ARM CPU Architecture

There are a number of considerations when porting Linux applications from x86_64 to ARM:

- The instruction set differs. In particular, assembly or compiler-intrinsics-based SIMD code needs to use the appropriate ARM extensions instead (see *CPU Instruction Extensions*).
- The memory ordering differs. Memory accesses between threads within the same process are weakly ordered on ARM, while x86_64 implements *total store order*. Correctly written multi-threaded programs behave the same on both architectures. But faulty code can result in unexpected behavior when porting to ARM. See also *ARM Memory Ordering*.

Chapter 3. SIMD

NVIDIA DGX Spark implements two vector single-instruction-multiple-data (SIMD) instruction extensions:

- Advanced SIMD Instructions (NEON)
- ARM Scalable Vector Extensions (SVE)

ARM Advanced SIMD Instructions (or NEON) is the most common SIMD ISA for ARM64. It is a fixed-length SIMD ISA that supports 128-bit vectors. The first ARM-based supercomputer to appear on the Top500 Supercomputers list (Astra) used NEON to accelerate linear algebra, and many applications and libraries are already taking advantage of NEON.

More recently, ARM64 CPUs have started supporting ARM Scalable Vector Extensions (SVE), which is a length-agnostic SIMD ISA that supports more datatypes than NEON (for example, FP16), offers more powerful instructions (for example, gather/scatter), and supports vector lengths of more than 128 bits. SVE is currently found in NVIDIA DGX Spark, NVIDIA Grace, the AWS Graviton 3, Fujitsu A64FX, and others. SVE is not a new version of NEON, but an entirely new SIMD ISA.

NVIDIA DGX Spark can retire six 128-bit NEON operations or six 128-bit SVE2 operations on P-cores, and two 128-bit NEON operations or two 128-bit SVE2 operations on E-cores. Although the theoretical peak performance of SVE and NEON are the same for these CPUs, SVE (and especially SVE2) is a more capable SIMD ISA with support for complex data types and advanced features that enable the vectorization of complicated code. In practice, kernels that cannot be vectorized in NEON can be vectorized with SVE. So, although SVE will not beat NEON in a performance drag race, it can dramatically improve the overall performance of the application by vectorizing loops that would have otherwise executed with scalar instructions.

Chapter 4. Porting Existing Apps

Table of Contents:

4.1. Dependencies & Tools

The following table lists NVIDIA software packages with Spark support status, minimum supported version and a link to the latest supported release. Note that support might extend over time, so make sure to always check the latest version of this porting guide. Last Updated: February 13, 2026.

Package	Supported	Min. Version	Notes
AI Aerial	No	n/a	
Compute Sanitizer	Yes	n/a	Included in *CUDA Toolkit*
Container Toolkit	Yes	1.17.7	
cuCIM	No	n/a	
cuBLAS	Yes	n/a	Included in *HPC SDK* and *CUDA Toolkit*
CUDA Toolkit	Yes	13.0	
CUDA-GDB	Yes	n/a	Included in *CUDA Toolkit*
CUDA Math API	Yes	n/a	Included in *CUDA Toolkit*
cuDNN	Yes	9.11	
cuDSS	Yes	0.7.0	
cuEquivariance	No	n/a	
cuFFT	Yes	n/a	Included in *HPC SDK* and *CUDA Toolkit*
cuOpt	Yes	25.12	
CUPTI	Yes	n/a	Included in *CUDA Toolkit*
cuPyNumeric	Yes	26.01	
cuQuantum	Yes	25.06	
cuRAND	Yes	n/a	Included in *HPC SDK* and *CUDA Toolkit*

continues on next page

Table 1 – continued from previous page

Package	Supported	Min. Version	Notes
cuSolver	Yes	n/a	Included in *HPC SDK* and *CUDA Toolkit*
cuSPARSE	Yes	n/a	Included in *CUDA Toolkit*
cuTensor	Yes	2.3	
CUTLASS	Yes	4.2.0	C++ Supported
cuVS	Yes	25.12	
CV-CUDA	No	n/a	
DeepStream	Yes	8.0	
HPC SDK	No	n/a	
Deep Learning Frameworks	Yes	25.09	
DOCA	No	n/a	
MONAI	Yes	TBD	
NCCL	Yes	2.28.9-1	
NPP	Yes	n/a	Included in *CUDA Toolkit*
Nsight Aftermath SDK	No	n/a	
Nsight Cloud	No	n/a	
Nsight Compute	Yes	n/a	Included in *CUDA Toolkit*
Nsight Deep Learning Designer	No	n/a	
Nsight Graphics	No	n/a	
Nsight Perf SDK	No	n/a	
Nsight Systems	Yes	TBD	
Nsight Tools JupyterLab Extension	No	n/a	
Nsight Visual Studio Code Edition	Yes	2025.1	
nvImageCodec	No	n/a	
nvJPEG2000	No	n/a	
nvmath-python	Yes	0.8	
NVRTC	Yes	n/a	Included in *CUDA Toolkit*
nvTIFF	No	n/a	
NVTX	Yes	n/a	
OptiX	Yes	9.0	
RAPIDS	Yes	25.10	

continues on next page

Table 1 – continued from previous page

Package	Supported	Min. Version	Notes
Streamline	No	n/a	
TensorRT	Yes	10.14.1	
TensorRT for RTX	No	n/a	
TensorRT Model Optimizer	No	n/a	
TensorRT-LLM	Yes	1.2	
Triton	Yes	25.08	
Warp	Yes	1.10.0	

4.2. DGX Spark Software Stack

This document outlines the software specifications for the NVIDIA DGX Spark product, providing an overview of its operating system and core software stack. The DGX Spark is designed for software developers and AI enthusiasts looking to leverage the DGX ecosystem for local AI development.

4.2.1. Overview

The DGX Spark offers a robust and versatile software environment tailored for advanced AI applications. It features a base operating system derived from Ubuntu 24.04 and integrates the NVIDIA AI stack, providing access to essential tools and libraries for AI and machine learning workflows.

4.2.2. System Architecture

4.2.2.1 Operating System

- **Base OS**: Ubuntu 24.04 server image with desktop packages
- **Kernel**: Linux v6.11 based NVIDIA Base OS with necessary patches

4.2.2.2 Boot and Hardware Enablement

Boot Configuration

- **Boot Mode**: UEFI (default), with USB-based boot support
- **Initial Setup**: Configurable system settings on first boot:
  - Timezone, language, keyboard layout
  - Username, password, and hostname

Operational Modes

- **Desktop Mode**: Standard operation with display, keyboard, and mouse
- **Headless Mode**: Network-accessible via SSH, webserver

Firmware and Updates

- **BSP Firmware**: NVIDIA-supported with independent OS updates
- **Update Methods**: Secure UEFI capsule updates and LVFS (Linux Vendor Firmware Service)
- **System Updates**: Repository-based over-the-air (OTA) updates

Hardware Support

- **Storage**: Single internal NVMe SSD (M.2 form factor)
  - Capacity: 1TB-4TB
  - Features: SED-based hardware encryption
- **System Memory**: 128 GB unified memory
- **GPU Driver**: Latest iGPU driver optimized for AI stack
  - Launch Driver: R580.GA UDA driver
- **USB Support**: USB 3.2 driver support for:
  - Baseline devices
  - HID devices
  - Webcams

Networking

- **Ethernet**: Full support with MLNX_OFED drivers for CX7 (Connect7 PCIe-based smart NIC)
- **Wireless**: WiFi and Bluetooth drivers included
- **Bluetooth Profiles**: BT HID profiles at launch, broader profiles (BT Audio, BLE) post-launch

System Recovery

- **Re-imaging**: USB-based system recovery
- **Image Sources**: Canonical/NVIDIA repositories

Security

- **Boot Security**: dTPM (Discrete Trusted Platform Module)
  - Default: Off for first-party DGX Spark systems
  - Configurable: Can be enabled by enterprises/OEMs with signed driver/kernel

4.2.3. Display and Desktop Interface

4.2.3.1 Display Capabilities

Video Outputs

- **HDMI**: 1x HDMI 4K (up to 120Hz)
- **DisplayPort**: 2x DP (Alt mode) 4K (up to 120Hz)

Audio Support

- USB Audio
- Bluetooth Audio

4.2.3.2 Desktop Experience

- **Interface**: Regular Ubuntu desktop with NVIDIA branding
- **Pre-installed**: Default icons for NVIDIA software, documentation, and how-to videos
- **Graphics**: Ubuntu (Wayland) GUI desktop with preinstalled browser
- **Acceleration**: Desktop and application acceleration using OpenGL/Vulkan
- **Video**: Desktop video acceleration (nvenc/nvdec) for browsers and media players (VLC)

DRM Content Support

- Browser playback in fallback resolutions
- Enhanced copy protection (planned post-launch)

4.2.3.3 Performance and Power Management

- **RTD3**: Runtime D3 support
- **Power States**: Product-defined PStates for optimized performance
- **Suspend/Resume**: Basic functionality support

4.2.4. Software Stack

4.2.4.1 Core AI Libraries

NVIDIA AI Stack

- NCCL (NVIDIA Collective Communications Library)
- cuDNN (CUDA Deep Neural Network library)
- TensorRT-LLM
- TensorRT
- All supported toolkits and math libraries

CUDA Toolkit

- CUDA 13.0
- Latest fully-tested CUDA Toolkit, with CUDA examples included

4.2.4.2 Development Tools

Linux Development Tools

- build-essentials
- gdb, vim
- Support for C, C++, Perl, Python development

GPU Development Tools

- Nsight Systems
- Nsight Compute
- Nsight Graphics

- Nsight Deep Learning Designer
- JupyterLab extensions
- CUDA GDB

4.2.4.3 Container and Orchestration

Docker Support

- NVIDIA Docker containers
- NVIDIA Container Runtime for Docker included
- Multiple bare metal container support

Kubernetes

- Single and stacked device support

4.2.4.4 Data Science and Analytics

Python Data Stack (PyData)

- cuDF
- cuML
- cuGraph
- XGBoost

Apache Spark / Spark RAPIDS

- Enterprise data science support
- RAPIDS OSS project support for CUDA 13.0

Deep Learning Frameworks

- All Blackwell-optimized frameworks provided by NVIDIA

Compute Support

- OpenCL support included

4.2.4.5 Additional Software Support

- **Jetson SW**: Jetson software services on SBSA CUDA
- **Omniverse**: NVIDIA Omniverse support
- **GPU Driver**: GSP-RM/OpenRM kernel module (default)
- **Telemetry**: Device activation and census support

4.2.5. System Management

4.2.5.1 Monitoring and Diagnostics

System Monitoring

- nvidia-smi for basic system health monitoring
- GPU and CPU telemetry via system monitoring agents

- MiTelemetry support for system controllers

Hardware Diagnostics

- Hardware error recording (FDR) for error history
- Field diagnostics software for RMA flow management
- Manufacturing diagnostics with external SOC support for MODS
- CPU and GPU testing capabilities

Remote Management

- Serial console support for flashing and remote management

4.2.6. Security Features

Secure Boot and TPM

- Secure boot and TPM (Trusted Platform Module) support
- Default: Off for TPM systems
- Configurable: Can be enabled by enterprises/OEMs with signed driver/kernel

Firmware Security

- Signing infrastructure for all firmware/BSP components
- Secure firmware updates via EC (Embedded Controller) and UEFI

4.2.7. Performance Specifications

4.2.7.1 Chip Features

CPU Configuration

- **Architecture**: 10P+10E
- **All-cores FMax**:
  - PCores: 4.075GHz
  - ECores: 2.8GHz (50% Bin)
- **Turbo/Single-core FMax**:
  - PCores: 4.175GHz
  - ECores: N/A (50% Bin)
- **PCore VMax**: 1.2V

Feature Support Matrix

Table 2: Feature Support Matrix

Feature	Status
ISP	NO
DLA	NO
Audio/Audio DSP	No external Codec
OSROOT/FTPM	No (External TPM)
Sensor Hub	No
dGPU attach	No
10s	No
DP over USB4	Enabled post-launch
CSI	NO
eDP	NO
Soundwire	NO

4.2.7.2 Reliability Specifications

Yield and Lifetime

- **Yield/Bin Size**: 50%/Typical (No corner part characterization needed)
- **Lifetime at Vmax**: 25% of 5 Years at 105°C (~70°C ambient) max perf state (180 hrs/pm, TJMAX)
- **Lifetime at Nominal Voltage**: 75% of 5 Years at 55°C TJMAX (35°C ambient) (540 hrs/pm at Vmin or suspend)

Reliability Metrics

- **EM (Electromigration)**: 1000/10yr (Commercial segment)
- **Design DPPM (intrinsic)**: AGING: 500 dppm total (with margins)
- **Design Target**: Median 0.75yr at Vmax/105°C
- **TDDB (Time Dependent Dielectric Breakdown)**: 500-600

4.3. Setup and Build Systems

4.3.1. Compilation on DGX Spark

4.3.1.1 CMake

CMake can automatically detect CUDA toolchains when certain prerequisites are met. Since DGX Spark comes with the CUDA SDK pre-installed, CMake will detect and use it automatically. For example, run the following on your DGX Spark:

```
git clone https://github.com/NVIDIA/CUDALibrarySamples.git
cd CUDALibrarySamples/cuBLAS/Level-3/gemm
```

(continues on next page)

(continued from previous page)

```
mkdir build
cd build
cmake  -DCMAKE_CUDA_ARCHITECTURES="121-real" ..
cmake --build .
./cublas_gemm_example
```

If cmake is unable to find nvcc, add `-DCMAKE_CUDA_COMPILER=/usr/local/cuda/bin/nvcc` to the first `cmake` call. For optimal performance, make sure to compile for the compute capability of your device. For DGX Spark, this is 121-real.

4.3.2. Cross-compilation on x86_64 Linux host

In the context of this guide, cross-compilation refers to compiling applications on x86_64 Linux devices (the host) for the arm64-based DGX Spark (the target). To build and link correctly, you need to install arm binaries and modify the toolchain on your host device. For simplicity and to avoid unintended side effects, we demonstrate cross-compilation using Docker. As our example project, we use the `cuda-samples` project, available on github.

Please use the newest available toolkit, starting from version 13.0.

Build steps:

1. Set up Docker and verify your Docker installation with `docker run hello-world`
2. Create a basic Dockerfile:

```
# Use the NVIDIA CUDA base image with cuDNN and Ubuntu 24.04
FROM nvidia/cuda:13.0.0-cudnn-devel-ubuntu24.04

# Set environment variables to avoid prompts during package installation
ENV DEBIAN_FRONTEND=noninteractive

# Update and install basic utilities
# python3 is needed to install and configure cuda-cross-sbsa for cross-
↪compilation
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        build-essential wget curl ca-certificates git python3 python3-pip
↪python3-venv && \
    rm -rf /var/lib/apt/lists/*

RUN apt update
# Install build tools and arm64 toolchain
RUN apt install -y cmake gcc-aarch64-linux-gnu g++-aarch64-linux-gnu
# CUDA toolkit download: https://developer.nvidia.com/cuda-downloads:
↪Select linux->arm64-sbsa->Cross->Ubuntu->24.04->deb(network)
# Adjust installation instructions below as necessary
RUN wget https://developer.download.nvidia.com/compute/cuda/repos/
↪ubuntu2404/cross-linux-sbsa/cuda-keyring_1.1-1_all.deb \
    && dpkg -i cuda-keyring_1.1-1_all.deb \
    && apt update \
    && apt install -y cuda-cross-sbsa
```

(continues on next page)

(continued from previous page)

```
# Clone the repository
RUN git clone https://github.com/NVIDIA/cuda-samples
```

3. Build and launch the Docker container (adjust paths as needed):

```
docker build -t cross_compile_docker .
docker run -it --rm --runtime=nvidia --gpus 'all' -v ./host_dir:/docker_
↪dir cross_compile_docker /bin/bash
```

4. Navigate to `cuda-samples/Samples/0_Introduction/vectorAdd/`
5. Configure the build:

```
cmake -S . -B . -DCMAKE_CXX_COMPILER=/usr/bin/aarch64-linux-gnu-g++ -
↪DCMAKE_CUDA_COMPILER=/usr/local/cuda/bin/nvcc -DCMAKE_CUDA_HOST_
↪COMPILER=/usr/bin/aarch64-linux-gnu-g++
```

6. Build the project:

```
cmake --build .
```

   Note that warnings about missing CUDA runtimes might appear.

7. Either move the `vectorAdd` binary to your Spark, or copy the Docker container itself. The DGX Spark should be able to run the binary in both cases. If you encounter permission issues when running the binary on the Spark, use `chmod +x <binary_name>` to make the binary executable on the ARM64 host.
8. Alternatively, you can check the binary information using `file vectorAdd`. The output should be similar to:

```
vectorAdd: ELF 64-bit LSB pie executable, ARM aarch64, version 1 (SYSV),
↪dynamically linked, interpreter /lib/ld-linux-aarch64.so.1,
↪BuildID[sha1]=df539f8a4bbaeeadadcd9b816be6ea230ed72d42, for GNU/Linux 3.
↪7.0, not stripped
```

4.4. CPU Instruction Extensions

4.4.1. AVX2 and AVX-512

The libraries SIMDe and SSE2NEON are designed as drop-in replacements for x86_64 SIMD instruction set extensions like SSE and AVX, through emulation via ARM NEON and SVE. These libraries provide a good starting point for porting existing x86_64 SIMD code to ARM.

With the newer ARM extensions SVE and SVE2, many x86_64 intrinsics that have no direct NEON counterpart and therefore require emulation via multiple instructions now have straightforward replacements. We recommend porting them directly to SVE. This applies mainly to gather, scatter, predicated, and SM4 encryption instructions.

ARM v9.2 also supports the bit extension `SVE_BitPerm`, which introduces BEXT, BDEP, and BGRP. BDEP is an alternative to PDEP, part of the Intel BMI2 extension.

When using SVE, there is a choice between variable and fixed vector length and therefore register size. If you already use ASIMD/NEON or SIMDe and SSE2NEON, you should add SVE code on a case-by-case basis and fix the vector length to 128-bit for compatibility and performance. Mixing NEON and SVE incurs no additional overhead.

SVE supports both *vector-length agnostic* (VLA) and *vector-length specific* (VLS) programming. VLA is portable, so it works on hardware with any vector register bit-width. VLS sacrifices this portability to make the vector width known to the compiler and therefore enables further optimizations.

4.4.1.1 VLA: Scalable Vectors

The vector length is not set by default and can be unset manually via the compiler flag `-msve-vector-bits=scalable`. Not knowing this can lead to performance degradation, because:

- The compiler may not unroll the loop.
- Partial use of the SVE register requires appropriate predication.
- Run-time checks are required to map AVX to SVE types and operations.

For example, C or C++ doesn't allow data structs to contain types of unknown size like `svuint32x2`. These run-time checks also reduce instruction level parallelism (ILP) and out-of-order (OoO) execution capacity.

```
#ifndef __ARM_FEATURE_SVE_BITS
#define __m256i svuint32x2_t

__m256i _mm256_i32gather_epi32(const int* base_addr, __m256i vindex, const
↪int scale) {
    __m256i result = svset2_s32(
        result,
        0,
        svld1_gather_s32offset_s32(
            svwhilelt_b8_s32(0, 32),
            base_addr,
            svmul_n_s32_x(svptrue_b32(), svget2_s32(vindex, 0), scale *
↪sizeof(int))
        )
    );
    if (svcntb() < 32) {
        result = svset2_s32(
            result,
            1,
            svld1_gather_s32offset_s32(
                svwhilelt_b8_s32(16, 32),
                base_addr,
                svmul_n_s32_x(svptrue_b32(), svget2_s32(vindex, 1), scale *
↪sizeof(int))
            )
        );
    }
    return result;
}
#endif
```

4.4.1.2 VLS: Fixed-Length Vectors

You can use the compiler flag -msve-vector-bits={} to fix the vector length. The pre-processor macro __ARM_FEATURE_SVE_BITS is then set to the specified value. Otherwise, it is not defined. Passing 128 makes the code compatible with all ARM SVE-enabled cores and allows the compiler to incorporate this knowledge into optimization decisions.

The following sections describe SVE implementations of common AVX2 and AVX-512 intrinsics.

4.4.1.3 AVX2 Gather and Scatter

The gather and scatter intrinsics of the format _mm[register size]_[index data type][gather/scatter]_[data type] should be changed to a pair of consecutive svuint32_t svld1sb_[gather/scatter]_[data type]offset_[offset data type]. The multiply operation can be removed and svld1_gather used instead if the scaling factor is known at compile time and is equal to sizeof(data_type). The svptrue_[data type] intrinsic function call should be used as an argument for the svbool_t pg parameter (see the code snippet below):

```
#if __ARM_FEATURE_SVE_BITS==128
typedef svint32_t __m128i __attribute__((arm_sve_vector_bits(128)));
typedef struct __m256i { __m128i val[2]; } __m256i;
#define AVX_REGSIZE_DQ (256/128)
#define AVX512_REGSIZE_DQ (512/128)

__m256i _mm256_i32gather_epi32(const int* base_addr, __m256i vindex, const
↪int scale)
{
    __m256i result;
    svbool_t ptrue = svptrue_b32();
    for (int i = 0; i < AVX_REGSIZE_DQ; i++)
    { // compiler unrolls the loop
        result.val[i] = svld1sb_gather_s32offset_s32(
            ptrue,
            base_addr,
            svmul_n_s32_x(ptrue, vindex.val[i], scale * sizeof(int))
        );
    }
    return result;
}
#endif
```

4.4.1.4 AVX2 Masked Gather and Scatter

Porting masked operations from AVX2 is straightforward; however, SVE requires an extra multiplication (to apply the scale factor), a compare operation (because AVX2 uses vector registers as masks, while SVE uses predicate registers), and a bitselect operation (to merge with the third vector src). The same approach is applicable to porting scatter instructions.

```
__m256i _mm256_mask_i32gather_epi32(__m256i src, int const* base_addr, __
↪m256i vindex, __m256i mask, const int scale)
{
    __m256i result;
    svbool_t ptrue = svptrue_b32();
    for (int i = 0; i <  AVX_REGSIZE_DQ; i++)
```

(continues on next page)

(continued from previous page)

```
    { // compiler unrolls the loop
        svbool_t pmask32 = svcmpne_n_s32(ptrue, mask.val[i], 0);
        result.val[i] = svsel_s32(
            pmask32,
            svld1_gather_s32offset_s32(
                pmask32,
                base_addr,
                svmul_n_s32_x(ptrue, vindex.val[i], scale * sizeof(int))
            ),
            src.val[i]
        );
    }
    return result;
}
```

4.4.1.5 AVX-512 __mmask and SVE svbool_t

In `svbool_t`, every bit is mapped to the corresponding byte of the lane of the scalable vector register, while one bit is mapped to the entire lane in the AVX-512 `__mmask`. For example, SVE requires 4 bits to map the `svbool_t` to a dword lane, and only the least significant bit out of the four will be used in the operation with predicate. Although BDEP (vector operation), an alternative to x86_64 PDEP (scalar operation), was introduced in SVE2 (FEAT_BitPerm), it only helps when converting the `__mmask` bitmask to bytemask and in the case where `reinterpret_cast` is not an option: `__mmask` can be converted to byte mask and then to `svbool_t` using the SVE compare operation. The following code splits `__mmask` into four `svbool_t` values to map to four 128-bit `svuint32_t` vectors:

```
#if REINTERPRET_CAST_IS_NOT_ALLOWED
static svbool_t convert_mmask16_to_svbool8(__mmask16 k)
{
    uint64_t bit2bytemask = 0x101010101010101;
    svuint8_t bitmask = svreinterpret_u8_u64(svdup_n_u64(k));
    svuint8_t bytemask = svreinterpret_u8_u64(svbdep_n_u64(
        svreinterpret_u64_u8(svtbl_u8(bitmask, svreinterpret_u8_u64(svdupq_n_
↪u64(0, 1)))),
        bit2bytemask)
    );
    svbool_t pmaskb8 = svcmpeq_n_u8(svptrue_b8(), bytemask, 1);
    return pmaskb8;
}
#endif
```

```
static inline void convert_mmask16_to_svbool32(__mmask16 k, pred_(&
↪pmaskb32)[4])
{
    #if REINTERPRET_CAST_IS_NOT_ALLOWED
        svbool_t pmaskb8 = convert_mmask16_to_svbool8(k);
    #else
        svbool_t pmaskb8 = *reinterpret_cast<svbool_t*>(&k);
    #endif
    svbool_t pmaskb16lo = svunpklo_b(pmaskb8);
    pmaskb32[0] = svunpklo_b(pmaskb16lo);
```

(continues on next page)

(continued from previous page)

```
    pmaskb32[1] = svunpkhi_b(pmaskb16lo);
    svbool_t pmaskb16hi = svunpkhi_b(pmaskb8);
    pmaskb32[2] = svunpklo_b(pmaskb16hi);
    pmaskb32[3] = svunpkhi_b(pmaskb16hi);
}
```

There is no intrinsic SVE MOV operation "because the ACLE intrinsic calls do not imply a particular register allocation and so the code generator must decide for itself when move instructions are required" source.

4.4.1.6 AVX-512 Masked Gather and Scatter

Porting the masked gather or scatter intrinsics from AVX-512 to SVE is similar to porting from AVX to SVE, but k-register types like `__mmask16` should be converted to `svbool32_t` first (using the function mentioned above).

```
__m512i _mm512_mask_i32gather_epi32(__m512i src, __mmask16 k, __m512i vindex,
↪int const* base_addr, const int scale)
{
    __m512i result;
    pred_t pmaskb32[4];
    convert_mmask16_to_svbool32(k, pmaskb32);
    svbool_t ptrue = svptrue_b32();
    for (int i = 0; i < AVX512_REGSIZE_DQ; i++)
    {
        result.val[i] = svsel_s32(
            pmaskb32[i],
            svld1_gather_s32offset_s32(
                pmaskb32[i],
                base_addr,
                svmul_n_s32_x(ptrue, vindex.val[i], scale * sizeof(int))
            ),
            src.val[i]
        );
    }
    return result;
}
```

The `reinterpret_cast` is used because there are no SVE mov intrinsics, as described above.

4.4.1.7 AVX-512 Other Masked Operations

Most masked AVX-512 operations benefit from direct translation to SVE. For example, the intrinsics of the format `_mm[register_size]_[mask/maskz]_[operation]_[data type]` can be directly translated to SVE alternatives, because of the predicated registers available for both AVX-512 and SVE. The porting snippet below is applicable to numerous arithmetic, logic, comparison, and bit processing AVX-512 intrinsics containing mask or maskz suffixes:

```
__m512i _mm512_maskz_add_epi32(__mmask16 k, __m512i a, __m512i b)
{
    __m512i result;
    pred_t pbmaskb32[4];
```

(continues on next page)

(continued from previous page)

```
    convert_mmask16_to_svbool32(k, pbmaskb32);
    for (int i = 0; i < AVX512_REGSIZE_DQ; i++)
    { // compiler unrolls the loop
        result.val[i] = svadd_s32_z(pbmaskb32[i], a.val[i], b.val[i]);
    }
    return result;
}
```

The same approach as above, but with mask-suffixed intrinsics, has a broader workaround because the SVE2 predicated operation can merge with the first operand of the SVE operation. However, porting to SVE2 still has advantages over NEON, particularly in converting the `__mmask16` bitmask type to a vector register with fewer instructions:

```
__m512i _mm512_mask_add_epi32(__m512i src, __mmask16 k, __m512i a, __m512i b)
{
    __m512i result;
    pred_t pmaskb32[4];
    convert_mmask16_to_svbool32(k, pmaskb32);
    for (int i = 0; i < AVX512_REGSIZE_DQ; i++)
    { // compiler unrolls the loop
        result.val[i] = svsel_s32(
            pmaskb32[i],
            svadd_s32_x(pmaskb32[i], a.val[i], b.val[i]),
            src.val[i]
        );
    }
    return result;
}
```

4.4.1.8 AVX-512 Reduce Operations

Replace instructions like `_mm_reduce_add_epi32` with SVE horizontal reduce intrinsics like `svaddv_s32()`. It is not recommended to rely on auto-vectorization for reductions.

4.4.1.9 SM4 and SM4KEY Encryption

NEON does not provide instructions similar to x86_64 encryption intrinsics like `_mm(_mm256)_sm4round4_epi32` and `_mm(_mm256)_sm4key_epi32`. However, with SVE2, these encryption intrinsics can be directly replaced by `svsm4e_u32` and `svsm4ekey_u32` respectively.

4.5. ARM Memory Ordering

Memory ordering describes the order in which CPUs access memory and the order in which these accesses can be observed by other CPUs. If the compiler or CPU reorder memory accesses for a single threaded application, they ensure that the result of the program will not change. Multithreaded applications however must explicitly make use of memory barriers to ensure correct operation.

Code for x86_64 also must keep memory ordering in mind; however, some reorderings cannot happen on x86_64 but can happen on ARM (and in fact most other CPU architectures). This is why multi-threaded code being ported from x86_64 to another architecture can show race conditions and other multithreading related bugs that were not present on x86_64.

Explicit memory barriers are provided via C11 or C++11 atomics (e.g. atomic_thread_fence() from stdatomic.h).

NVIDIA DGX Spark has support for the Large-System Extension (LSE), which was first introduced in Armv8.1. LSE provides low-cost atomic operations that can improve system throughput for thread communication, locks, and mutexes. On recent ARM64 CPUs, the improvement can be up to an order of magnitude when using LSE atomics instead of load/store exclusives.

When building an application from source, the compiler needs to generate LSE atomic instructions for applications that use atomic operations. For example, the code of databases such as PostgreSQL contain atomic constructs: C++11 code with std::atomic statements that translate into atomic operations. Since GCC 9.4, GCC's -moutline-atomics or -march=armv8-a+lse flag enables LSE instructions (if -march=armv8.1-a or higher target architecture is used LSE will be included directly). To confirm that LSE instructions are created, the output of objdump command-line utility should contain LSE instructions:

```
$ objdump -d app | grep -i 'cas\|casp\|swp\|ldadd\|stadd\|ldclr\|stclr\|ldeor\
↪|steor\|ldset\|stset\|ldsmax\|stsmax\|ldsmin\|stsmin\|ldumax\|stumax\
↪|ldumin\|stumin' | wc -l
```

To check whether the application binary contains load and store exclusives, run the following command:

```
$ objdump -d app | grep -i 'ldxr\|ldaxr\|stxr\|stlxr' | wc -l
```

See also:

- https://developer.arm.com/documentation/102336/0100/Memory-ordering
- https://en.wikipedia.org/wiki/Memory_ordering

4.6. CUDA

4.6.1. GPUDirect RDMA

DGX Spark uses a unified memory architecture. On CUDA contexts the system memory returned by the pinned device memory allocators (e.g. `cudaMalloc`) cannot be coherently accessed by the CPU complex nor by I/O peripherals like PCI Express devices (e.g. the Mellanox NIC).

Hence the GPUDirect RDMA technology is not supported, and the mechanisms for direct I/O based on that technology, for example nvidia-peermem (for DOCA-Host), dma-buf or GDRCopy, do not work.

A compliant application should programmatically introspect the relevant platform capabilities, e.g. by querying `CU_DEVICE_ATTRIBUTE_GPU_DIRECT_RDMA_SUPPORTED` (related to nv-p2p kernel APIs) or `CU_DEVICE_ATTRIBUTE_DMA_BUF_SUPPORT` (related to dma-buf), and leverage an appropriate fallback.

For example, for Linux RDMA applications based on the ib verbs library, we suggest to allocate the communication buffers with the `cudaHostAlloc` API and to register them with the ib_reg_mr function.

Chapter 5. Optimization for DGX Spark

This section discusses software optimization practices that are relevant for DGX Spark.

5.1. General practices

- Profile your application to find hotspots that would benefit most from optimization. Making assumptions about hotspots can lead to incorrect conclusions. Some appropriate tools are discussed in a section below.
- Create reproducible microbenchmarks when optimizing hotspots. Measuring the runtime of short functions can be tricky; consider using a framework like google benchmark or nanobench. These frameworks generally handle the most notorious microbenchmarking challenges such as data variance, outliers, warmups, and subtracting overhead. Setting up microbenchmarks will help you track your progress while optimizing a hotspot.
- If possible, take advantage of vectorization, either by using your compiler's auto-vectorization or doing it by hand. SIMD is generally faster and more power-efficient with respect to work done than scalar code. If you are migrating vectorized x86_64 code, see *CPU Instruction Extensions*.
- When parallelizing large workloads using OpenMP, also consider `guided` or `dynamic` scheduling to better balance tasks in a hybrid architecture. If tasks complete quickly, try increasing the chunk size to keep synchronization overhead under control.
- Pay attention to memory access patterns. If appropriate, use tiling methods to maintain data locality. Note that DGX Spark is a hybrid architecture with varying cache sizes across its cores.
- The cores in DGX Spark are organized as two clusters with up to 5 P-cores and 5 E-cores each. Minimize concurrent writes to the same cache lines by different cores, especially ones in different clusters. Cache lines are 64 bytes long. Keep thread synchronization to a necessary minimum.

5.2. Compiler configuration

DGX Spark is based on ARMv9.2-A architecture. Developers may choose to target an older architecture to maintain compatibility; it is recommended to target at least ARMv8.2 with the `rcpc` extension.

5.2.1. gcc and clang

A specific architecture can be targeted by adding the `-march` switch, e.g. `-march=armv8.2-a`. When compiling on the Spark itself, `-march=native` will have the compiler target the host machine's architecture. Otherwise, `-march=armv9.2-a` can be used to specify Spark's ARMv9.2-A architecture.

Since LLVM version 21 and gcc version 15, it is also possible to specifically target DGX Spark with `-mcpu=gb10`. This enables optimizations for the Cortex-X925 and Cortex-A725 cores as well as the optional crypto extension and is thus preferred over a more generic `-march` switch. When compiling on the Spark itself, this architecture is automatically selected with `-march=native`.

When targeting an architecture older than ARMv8.2-A, it is advisable to also enable `-moutline-atomics`. This option will replace inline atomics with calls to the best atomics available for the platform, as determined at runtime.

5.3. .NET applications

ARM Advanced FP & SIMD (a.k.a. NEON) data types are available in modern .NET runtimes through the *Vector<T>*, *Vector128<T>*, and similar classes. Operations and ISA extensions on these vectors are accessible in the System.Runtime.Intrinsics.Arm namespace. When targeting multiple architectures, the availability of a particular ISA extension can be determined at runtime, e.g. `System.Runtime.Intrinsics.Arm.Dp.IsSupported`.

Access to SVE instructions is currently experimental and may have implications when using NativeAOT because of the runtime-determined vector width. See dotnet issue 93095 and Engineering the Scalable Vector Extension in .NET for more information.

5.4. Programming CUDA on UMA systems

To efficiently use the DGX Spark consider reading the CUDA Programming Guide. The guide provides detailed information regarding platform-specific considerations and best practices for optimizing CUDA workflows on UMA-enabled hardware.

5.5. Memory reporting on UMA systems (including DGX Spark)

DGX Spark systems use a unified memory architecture (UMA), where the GPU shares system memory (DRAM) with the CPU and other compute engines. This design reduces latency and allows larger amounts of memory to be used for GPU workloads. On UMA systems, the CPU can dynamically manage DRAM contents, including freeing up memory from the system's SWAP area and page caches. However, the `cudaMemGetInfo` API does not account for memory that could potentially be reclaimed. As a result, the memory size reported by `cudaMemGetInfo` may be smaller than the actual allocatable memory, since the CPU may be able to release additional DRAM pages.

To more accurately estimate the amount of allocatable device memory on DGX Spark platforms, CUDA application developers should consider the memory that can be reclaimed from the OS and not rely solely on the values returned by `cudaMemGetInfo`. For a reference implementation using C standard libraries, see the link below. This snippet returns the maximum memory that can be allocated without swapping (`availableMemory`) and the maximum memory that can be allocated with swapping (`availableMemory + freeSwap`).

See Guidance for reporting memory resources with unified memory architecture.

As a workaround for debugging purposes, you can flush the buffer cache manually with the following command:

```
sudo sh -c 'sync; echo 3 > /proc/sys/vm/drop_caches'
```

After flushing the cache, restart your application.

5.6. Profiling

Profiling your application is highly advisable before undertaking any optimization efforts. The tools of choice on Linux platforms are Nsight tools and `perf`.

5.6.1. Nsight Systems

NVIDIA Nsight Systems is a powerful performance analysis tool that presents the captured traces in a top-down graphical manner. Profiles are presented as a timeline of evolving metrics, such as per-thread CPU usage, GPU metrics and other data.

It is possible to profile an application through an SSH connection using Nsight systems. When creating an Nsight project, select an existing SSH connection or add a new one using the *Configure targets...* item.

Additionally NVIDIA Nsight Graphics and NVIDIA Nsight Compute are available to analyze 3D rendering and compute workloads.

5.6.2. `perf`

Recording perf traces has security implications; you may need to unlock access to performance counters by adding a file to `/etc/sysctl.d` with the following content:

```
kernel.perf_event_paranoid=-1
kernel.kptr_restrict=0
```

It is also possible to enable these settings only until the next reboot. To install the perf tools, execute:

```
sudo apt install linux-tools-$(uname -r)
```

To record a trace for your application and launch the analyzer:

```
sudo perf record -e cycles,(...) ./your-application
perf report
```

In `perf report`, a top-down view of hot functions is presented. A mixed C++ and disassembler view with proportion of time spent can be generated by selecting a function (shared object) and using the Annotate function. If the C++ code is not shown in the disassembler view, ensure the source code can be found by the profiler and that the application was compiled with the `-g` switch, which includes debug information in the binary.

For a listing of available performance monitoring counters, see Arm Cortex-X925 Performance monitoring events and Arm Cortex-A725 Performance monitoring events.

5.7. Links

- A64 SIMD Instruction List: SVE Instructions

Chapter 6. Feedback

If you have feedback on this guide or questions not covered herein, please contact your developer relations person or account manager.

You can also visit the DGX Spark developer forums: https://forums.developer.nvidia.com/c/accelerated-computing/dgx-spark-gb10/719

Copyright

©2022-2026, NVIDIA Corporation