

LIBXSTREAM Samples

Ozaki Scheme – OpenCL

High-precision GEMM on OpenCL devices via mantissa slicing (Scheme 1) or Chinese Remainder Theorem (Scheme 2). Both schemes decompose FP matrices into int8/u8 tiles and use DPAS/XMX matrix engines when available. This is an OpenCL adaptation of the CPU-based Ozaki sample in LIBXS.

Build

```
cd samples/ozaki
make [GNU=1] [DBG=1]
```

Requires an OpenCL runtime and headers. BLAS is linked via `BLAS=2` for the reference GEMM.

Run

```
./ozaki.x [M [N [K [transa [transb [alpha [beta [lda [ldb [ldc]]]]]]]]]]]
```

All arguments are positional and optional:

Pos.	Argument	Default	Description		1	M	257	Rows of C and op(A)	2	N	M	Columns of C and op(B)	3	K	M	Inner dimension	4	transa	0	0=N, 1=T for A	5	transb	0	0=N, 1=T for B	6	alpha	1	Scalar multiplier for A*B	7	beta	1	Scalar multiplier for C	8	lda	auto	Leading dimension of A	9	ldb	auto	Leading dimension of B	10	ldc	M	Leading dimension of C	
------	----------	---------	-------------	--	---	---	-----	---------------------	---	---	---	------------------------	---	---	---	-----------------	---	--------	---	----------------	---	--------	---	----------------	---	-------	---	---------------------------	---	------	---	-------------------------	---	-----	------	------------------------	---	-----	------	------------------------	----	-----	---	------------------------	--

Environment Variables

Scheme Selection	Variable	Default	Description		OZAKI	2	1=mantissa slicing, 2=CRT (default), 3=adaptive, 0=bypass BLAS	OZAKI_FP	64	64=fp64 (double), 32=fp32 (float)	OZAKI_N	(auto)	Slices (Sch.1: fp64=8, fp32=4) or primes (Sch.2: fp64=16, fp32=9)	
-------------------------	----------	---------	-------------	--	-------	---	--	----------	----	-----------------------------------	---------	--------	---	--

OZAKI=3 (adaptive) starts with Scheme 1 on the first call to learn the effective cutoff from preprocessing occupancy data. Subsequent calls compare the Scheme-1 pair count against the Scheme-2 prime count and pick the cheaper path. The cutoff is cached alongside the preprocessed buffers and reused on cache hits without any device-to- host readback.

Accuracy	Variable	Default	Description		OZAKI_FLAGS	3	Sch.1 bitmask: 1=Triangular, 2=Symmetrize, 0=full S^2. No Sch.2	OZAKI_TRIM	0	Precision levels to trim (0=exact). ~7 bits (Sch.1), ~4 bits (Sch.2)	OZAKI_I8	0	Sch.2: use signed i8 residues (moduli<=128) instead of u8	OZAKI_GROUPS	0	Sch.2: K-grouping factor, consecutive K panels share reconstr.	
-----------------	----------	---------	-------------	--	-------------	---	---	------------	---	--	----------	---	---	--------------	---	--	--

Hardware Control	Variable	Default	Description		OZAKI_RTM	(auto)	Register tiling M (power of two). Auto: 2 (HIER), 4 (256-GRF)	OZAKI_RTN	(auto)	Register tiling N (power of two). Auto: 2 (Intel GPU), 1 (other)	OZAKI_WG	0	Work-group size hint (0=no hint)	OZAKI_SG	(auto)	Sub-group size (forced to 16 with XMX)	OZAKI_BIGGRF	(auto)	Override 256-GRF detection (0=off, 1=on). HIER defaults to 128	OZAKI_KU	2	K-loop unroll factor	OZAKI_RC	8	DPAS repeat count (8 or 4)	OZAKI_PB	1	Sch.2: CRT prime batching factor	OZAKI_HIER	(auto)	Sch.2: hierarchical CRT (default on). Two-level Garner reconstr.	OZAKI_PREFETCH	0	Sch.1: enable prefetching	OZAKI_SCALAR_ACC	0	Sch.1: force scalar accumulation	
-------------------------	----------	---------	-------------	--	-----------	--------	---	-----------	--------	--	----------	---	----------------------------------	----------	--------	--	--------------	--------	--	----------	---	----------------------	----------	---	----------------------------	----------	---	----------------------------------	------------	--------	--	----------------	---	---------------------------	------------------	---	----------------------------------	--

Memory and Caching	Variable	Default	Description		OZAKI_DEVPOOL	0	Device memory pool via USM/SVM (eliminates per-call alloc overhead)	OZAKI_CACHE	0	Preprocessing cache bitmask: 1=A, 2=B, 3=both. Skips on match	
---------------------------	----------	---------	-------------	--	---------------	---	---	-------------	---	---	--

The preprocessing cache also stores the last effective cutoff from Scheme 1 occupancy detection. On cache hits the cutoff is reused without device-to-host readback, eliminating the sync bubble.

Benchmark	Variable	Default	Description		NREPEAT	1	Number of benchmark repetitions
OZAKI_VERBOSE	0	0=silent, 1=errors, 2=warnings, 3+=all. Neg.=all					

Additional variables for profiling, accuracy monitoring, and complex GEMM dispatch (OZAKI_PROFILE, OZAKI_THRESHOLD, OZAKI_STAT, OZAKI_EPS, OZAKI_RSQ, OZAKI_EXIT, OZAKI_COMPLEX) are handled by the LIBXS Ozaki sample (LIBXS), which owns the GEMM interceptor. See its README for details.

Kernel Registry

Scheme 1 fused GEMM kernels are compiled on demand via a JIT registry. The compile-time cutoff (OZAKI_CUTOFF) is baked into each kernel specialization, allowing the compiler to eliminate dead slice-pair iterations and reduce register pressure. The first call with a given cutoff value triggers JIT compilation (~100 ms); subsequent calls hit the registry cache. Typical workloads produce 2-3 specializations (full cutoff, reduced cutoff, each with/without bounds checking).

Example

```
./ozaki.x 256
```

Scheme 2 on a large matrix:

```
OZAKI=2 ./ozaki.x 4096
```

Adaptive scheme selection with caching:

```
OZAKI=3 OZAKI_CACHE=3 ./ozaki.x 4096
```

Quick Tuning Guide

Scheme 2 (CRT, OZAKI=2, default): fixed cost of P integer GEMMs plus hierarchical Garner reconstruction. Predictable performance regardless of data distribution. Use OZAKI_GROUPS for K-grouping at large sizes. The hierarchical CRT (OZAKI_HIER, on by default) halves private residue arrays and enables GRF128 for doubled thread occupancy.

Scheme 1 (mantissa slicing, OZAKI=1): up to $S*(S+1)/2$ integer GEMMs, but adaptive cutoff can reduce this substantially for narrow exponent spans. Use OZAKI_TRIM to trade accuracy for speed.

Adaptive (OZAKI=3): automatically picks the cheaper scheme per call based on preprocessing occupancy. Best with OZAKI_CACHE=3 to avoid repeated occupancy readbacks.

Enable OZAKI_CACHE=3 when A or B stays constant across calls. Enable OZAKI_DEVPOOL=1 for repeated calls with similar sizes.

Small Matrix Multiplication (SMM) – OpenCL

Batched small matrix multiplications (SMM) on OpenCL devices via the ACC LIBSMM interface. Originated from the DBCSR OpenCL backend, adapted to run on top of LIBXSTREAM. Includes GPU-accelerated kernels, a benchmark driver, and an auto-tuning framework.

Build

```
cd samples/smm
make [WITH_GPU=<device>] [ELEM_TYPE=<type>]
```

Produces acc_bench.x. Requires an OpenCL runtime, BLAS (BLAS=2), and LIBXS built from a sibling directory.

Make variable	Default	Description		ELEM_TYPE	double	Element precision: double or float	WITH_GPU
auto	Device for tuned parameters (PVC, A100, ...)	Fallback: all CSVs					

Benchmark Driver

```
./acc_bench.x [nrepeat [batchsize [M [N [K [nc [na [nb]]]]]]]]]
```

The kernel shape can also be given as $M \times N \times K$:

```
./acc_bench.x 5 30000 13x5x7
```

The first argument can be a file with one set of parameters per line. The batchsize argument accepts K/M/G suffixes for memory-budget mode.

Argument	Default	Description		nrepeat	66 (3+CHECK)	Number of timed repetitions	batchsize	30000
Matrix products per batch (stack size)	M	23	Rows of A and C	N	M	Columns of B and C	K	M
Columns of A / rows of B	nc	batchsize/16	Number of unique C-matrices	na	10nc	<i>Number of unique A-matrices</i> / / nb / 10nc		
			Number of unique B-matrices					

Environment Variables (Benchmark)	Variable	Default	Description		CHECK	-1	Accuracy: negative=auto-threshold, 0=off, positive=custom
	CHECK_H2D	-	Minimum H2D bandwidth (GB/s); fail if below				
	CHECK_DEV	-	Minimum device GFLOPS/s; fail if below		CHECK_HST	-	Minimum host GFLOPS/s; fail if below
	DEVICE	0	Device index (multi-rank: device = rank % ndevices)		NREPEAT_H2D	1	H2D copy repetitions for bandwidth measurement
	NREPEAT_SMM	1	SMM kernel launches per timed iteration (for profiling)		BATCHSIZE_SMM	-	Override batchsize (and optionally nrepeat: bs,nrep)

LIBSMM Kernel Parameters

OpenCL kernels are generated at runtime for any (M, N, K) shape within MAX_KERNEL_DIM. Tuned parameters can be embedded at build time or loaded at runtime for better performance.

Environment Variables (LIBSMM) Transpose kernel:

Variable	Default	Description		OPENCL_LIBSMM_TRANS_BUILDOPTS	-	Extra OpenCL build options
OPENCL_LIBSMM_TRANS_INPLACE	0	Non-zero: in-place transpose (no LDS)				
OPENCL_LIBSMM_TRANS_BM	auto	Block size in M-direction (0 < BM <= M)				

Multiply kernel:

Variable	Default	Description		OPENCL_LIBSMM_SMM_BUILDOPTS	-	Extra OpenCL build options
OPENCL_LIBSMM_SMM_PARAMS	auto	Tuned parameters: 0=disable, or CSV path				
OPENCL_LIBSMM_SMM_BS	auto	Intra-kernel mini-batchsize				
OPENCL_LIBSMM_SMM_BM	auto	Block size in M-direction (0 < BM <= M)				
OPENCL_LIBSMM_SMM_BN	auto	Block size in N-direction (0 < BN <= N)				
OPENCL_LIBSMM_SMM_AP	auto	Access pattern for parameter/stack array				
OPENCL_LIBSMM_SMM_AA	auto	Access pattern for A-matrix array				
OPENCL_LIBSMM_SMM_AB	auto	Access pattern for B-matrix array				
OPENCL_LIBSMM_SMM_AC	auto	Access pattern for C-matrix array				

The full list of tunable parameters is available via `tune_multiply.py --help`. Some parameters produce distinct code paths (e.g., SMM_BS=1 vs SMM_BS=2), so the parameter space is non-smooth.

Tuned Parameters

Pre-tuned parameter sets in `params/`:

File	Device		tune_multiply_PVC.csv	Intel Data Center GPU Max (PVC)		tune_multiply_BMG.csv	Intel Battlemage (BMG)
tune_multiply_A100.csv	NVIDIA A100		tune_multiply_H100.csv	NVIDIA H100			
tune_multiply_GH200.csv	NVIDIA GH200		tune_multiply_V100.csv	NVIDIA V100		tune_multiply_P100.csv	NVIDIA P100
tune_multiply_Mi250.csv	AMD MI250						

Parameters are matched by device ID at runtime with best-match fallback. Single and double precision coexist in the same CSV.

```
## Use embedded parameters (default)
./acc_bench.x 5 30000 13 5 7

## Disable tuned parameters
OPENCL_LIBSMM_SMM_PARAMS=0 ./acc_bench.x 5 30000 13 5 7

## Load from a specific CSV file
OPENCL_LIBSMM_SMM_PARAMS=params/tune_multiply_PVC.csv ./acc_bench.x 5 30000 13 5 7
```

Rebuild with a specific device's parameters embedded:

```
make realclean
make WITH_GPU=PVC
```

Auto Tuning

Uses OpenTuner to explore the parameter space per (M, N, K) kernel. The tuner communicates with the benchmark driver through environment variables.

```
cd samples/smm
pip install -r requirements.txt
```

Build `acc_bench.x` before tuning. Keep GPU clocks and driver state as stable as practical during a run.

tune_multiply.py Use `tune_multiply.py` when you want direct control over one tuning run or over JSON maintenance.

Tune one kernel and write JSON results in the current directory:

```
./tune_multiply.py 13x5x7
```

Limit the search time, choose the JSON directory, and set the benchmark batch size:

```
mkdir -p params/local
./tune_multiply.py 13x5x7 --stop-after=300 -p params/local -s 30000
```

Tune several explicit kernels from a file, one $M \times N \times K$ per line:

```
printf '%s\n' 13x5x7 23x23x23 32x32x32 > kernels.txt
./tune_multiply.py kernels.txt --stop-after=180 -p params/local
```

Merge JSON files into a CSV file:

```
./tune_multiply.py -m -p params/local -o tune_multiply_local.csv
```

Update stored device names after rebuilding for a different target:

```
./tune_multiply.py -u -p params/local
```

Check existing JSONs without re-tuning:

```
./tune_multiply.py -c -p params/local
```

Useful options:

Option	Meaning
--stop-after N	Stop the search after N seconds
-p path	Directory for JSON input and output
-s size	Benchmark batch size, also called stack size
-a level	Tuning level: 0=all, 1=most, 2=some, 3=least
-m	Merge JSON files into a CSV file
-o file	CSV output file
-u [device]	Update JSON device names
-c [epsilon]	Validate JSON entries
-d	Delete outperformed duplicates during merge

The tuner can run under MPI. It detects local MPI rank variables and uses them to select `LIBXSTREAM_DEVICE`, so ranks on the same node can use different devices. This is most useful when each rank is tuning a different kernel.

```
mpirun \
  ./tune_multiply.py 13x5x7 --stop-after=300 -p params/local : \
  ./tune_multiply.py 23x23x23 --stop-after=300 -p params/local
```

Interrupted tuning writes the best result seen so far. If you tune a different kernel with `tune_multiply.py` directly, remove any old OpenTuner database first:

```
rm -rf opentuner.db
```

The wrapper below does this cleanup automatically.

tune_multiply.sh Use `tune_multiply.sh` when you want to expand triplet specifications, retune a directory, or split a larger tuning job into parts.

Tune a compact Cartesian-product specification:

```
./tune_multiply.sh -t 300 4 10 15, 6 7 8, 23
```

Triplets are comma-separated groups. The command above expands to all $M \times N \times K$ combinations with M in 4 10 15, N in 6 7 8, and $K=23$.

Tune an explicit list instead:

```
./tune_multiply.sh -t 300 1x1x1 2x2x2 13x5x7 23x23x23
```

Split the same work into four parts and run the second part:

```
./tune_multiply.sh -t 300 -j 4 -i 2 4 10 15, 6 7 8, 23
```

Retune every JSON file found in a directory:

```
./tune_multiply.sh -u -p params/local -t 300
```

Limit the generated work before splitting it:

```
./tune_multiply.sh -t 180 -r 4 32 -m 64 -n 100 \
  4 8 16 32, 4 8 16 32, 4 8 16 32
```

Useful options:

| Option | Meaning | ||| | -t seconds | Time limit per kernel | | -p path | Directory for JSON files | | -s size | Benchmark batch size, also called stack size | | -a level | Tuning level: 0=all, 1=most, 2=some, 3=least | | -u | Retune JSON files found under -p | | -d | Ask the merge step to delete outperformed JSONs | | -c | Continue with the next kernel after an error | | -b | Tune triplets in reverse order | | -j parts | Total number of tuning parts | | -i index | Part to run, using 1-based numbering | | -r low high | Keep kernels with $\text{low} \leq MNK \leq \text{high}$ | | -m extent | Keep kernels with $M, N,$ and K no larger than this | | -n count | Keep only the first count kernels | | -f file | Read $M \times N \times K$ list from a file (one per line) | | -k id | Use a predefined triplet set |

Tuning from a File A text file with one $M \times N \times K$ per line can drive the tuning session. Lines starting with `#` are comments, and inline comments after `#` are stripped. Whitespace within an entry is ignored:

```
./tune_multiply.sh -t 300 -f retune_shapes.txt -p params/local
```

Under MPI, the file entries are partitioned across ranks as usual:

```
mpirun -np 8 ./tune_multiply.sh -t 300 -f retune_shapes.txt -p params/local
```

Bulk Tuning with MPI Under MPI, the wrapper defaults `-j` to the MPI world size and `-i` to `rank + 1`. It also forwards a normalized local rank to `tune_multiply.py`, so the Python tuner can select a different device per local rank.

For most MPI launchers, this is enough:

```
mpirun -np 8 ./tune_multiply.sh -t 300 -p params/local \
  4 10 15, 6 7 8, 23
```

You can still spell out parts explicitly when the launcher or scheduler needs it:

```
mpirun \
  ./tune_multiply.sh -t 300 -j 4 -i 1 -p params/local \
    4 10 15, 6 7 8, 23 : \
  ./tune_multiply.sh -t 300 -j 4 -i 2 -p params/local \
    4 10 15, 6 7 8, 23 : \
  ./tune_multiply.sh -t 300 -j 4 -i 3 -p params/local \
    4 10 15, 6 7 8, 23 : \
  ./tune_multiply.sh -t 300 -j 4 -i 4 -p params/local \
    4 10 15, 6 7 8, 23 \
>out.log 2>&1
```

To retune an existing JSON directory across eight MPI ranks:

```
mpirun -np 8 ./tune_multiply.sh -u -p params/local -t 300
```

Managing Tuned Parameters JSON files are the working format. CSV files are the deployable format. A typical retune flow is:

```
make realclean
make WITH_GPU=P100
mkdir -p params/p100
./tune_multiply.sh -u -p params/p100 -t 300
./tune_multiply.py -m -p params/p100 -o params/tune_multiply_P100.csv
```

Keep GPU driver state persistent during tuning (e.g., `nvidia-smi -pm ENABLED` on headless NVIDIA systems).

Stencil

Finite-difference stencil computation for seismic wave propagation (RTM, FWI) on GPUs. Three kernel paths are available:

- **FP32** (default): dedicated scalar kernel with SLM blocking
- **BF16-DPAS** (`STENCIL_BF16=1`): Dekker-split BF16 via hardware matrix units (DPAS/XMX)
- **INT8-DPAS** (`STENCIL_INT8=1`): Ozaki-1 INT8 with carried-forward exponent

The DPAS paths achieve single-precision accuracy from low-precision tensor operations.

Method

The 3D Laplacian is decomposed into three 1D operators applied along each axis (dimension splitting). Each 1D operator D is a $BLK \times BLK$ banded Toeplitz matrix (bandwidth $2R+1$) representing the FD stencil.

Parameters (compile-time):

```
BLK      = 32      block side length (32^3 output cube)
RADIUS   = 4       half-order (8th-order FD, 9-point stencil)
```

FP32 path (default) The dedicated FP32 kernel uses SLM tiling with a sliding window along the slow axis. Work-group size 32×8 , cooperative SLM fill, no DPAS dependency. Supports XYZ and ZYX memory layouts, PML absorbing boundaries, and compact/dispersion-fitted operator methods.

BF16 path (STENCIL_BF16=1) The wavefield block P and operator D are Dekker-split into BF16 digits. The matrix product $D \times P$ is computed as a sum of BF16 x BF16 DPAS calls that accumulate into FP32.

```
NDIGITS_A = 2      BF16 digits for operator (11-bit range)
NDIGITS_X = 3      BF16 digits for wavefield
```

Products per dimension: $NDIGITS_A \times NDIGITS_X = 6$ DPAS calls. Isotropic total: $3 \times 6 = 18$ DPAS calls per output block.

INT8 path (STENCIL_INT8=1) Ozaki-1 slicing: the FD operator fits in a single 7-bit signed digit (NSLICES_A=1). The wavefield is sliced into 1-3 digits at runtime depending on local dynamic range.

```
NSLICES_A = 1      INT8 digits for operator
NSLICES_X = 1-3    INT8 digits for wavefield (runtime adaptive)
K_PAD_I8  = 64     K dimension (k=32 DPAS alignment)
```

Products per dimension: $1 \times nslices_eff = 1-3$ DPAS calls. Isotropic total: $3 \times (1-3) = 3-9$ DPAS calls per output block.

A per-block carried-forward exponent (exp_buf) tracks the dynamic range across time steps. Double-buffered output-based tracking avoids race conditions between neighboring blocks.

2D Block I/O

The kernel exploits Intel 2D block load instructions for the A-side (operator) and sub-group block reads for the B-side (wavefield in SLM):

BF16 path:

```
A: intel_sub_group_2d_block_read_16b_8r16x1c
B: intel_sub_group_2d_block_read_transform_16b_16r16x1c (VNNI)
```

INT8 path:

```
A: intel_sub_group_2d_block_read_8b_8r32x1c (k=32)
B: intel_sub_group_block_read8 from row-major SLM
```

The operator D is stored dense (banded zeros included). The structural zeros cost no memory bandwidth (D fits in L1 after first access) and no effective compute (kernel is memory-bound on wavefield reads).

The INT8 kernel gathers the wavefield directly into SLM during the fill loop (no separate preprocess kernel).

Build

```
make GNU=1 DBG=1 PEDANTIC=2
```

Requires LIBXS (sibling directory ../../libxs) and an OpenCL 2.0 runtime with cl_intel_subgroup_2d_block_io and DPAS support.

Usage

```
./stencil.x [options]
```

```
-n <N>          grid dimension (NxNxN, default 256)
-nx/ny/nz <N>  individual grid dimensions
-t <steps>      time steps (default 100)
-d <dims>       operator terms: 3=isotropic, 9=TTI (default 3)
-h <spacing>    grid spacing in meters (default 10.0)
-v <model>      velocity: const | grad | layered | <file.bin>
-vmin <vel>     minimum velocity m/s (default 1500)
-vmax <vel>     maximum velocity m/s (default 4500)
-f <freq>       source frequency Hz (default 25)
-w <steps>      warmup steps excluded from timing (default 5)

-seg-salt       preset: SEG/EAGE Salt (676x676x210, h=20m)
-overthrust     preset: SEG/EAGE Overthrust (801x801x187, h=25m)
```

Performance is reported as GPoints/s (grid points updated per second).

Velocity Models

The driver supports loading external velocity models as flat binary files (float32, x-fastest order). Several standard models are publicly available for benchmarking.

Grid: 676 x 676 x 210 (n1=210 depth, n2=n3=676 lateral)
Spacing: 20 m
Range: 1500 - 4482 m/s
Size: ~385 MB (float32)
Notes: Salt body with sharp velocity contrasts.
Standard RTM benchmark (Aminzadeh et al., 1997).

Original (20 m): <https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Salt/fvp>

Resampled (40 m, 115x338x338, for cheaper runs): https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Salt/BENCH_27PT/v.bin

Reference wavefield (frequency-domain CBS solution): https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Salt/BENCH_27PT/cbs_salt_real.bin https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Salt/BENCH_27PT/cbs_salt_imag.bin

Grid: 801 x 801 x 187 (n1=187 depth, n2=n3=801 lateral)
Spacing: 25 m
Range: 2179 - 6000 m/s
Size: ~450 MB (float32)
Notes: Layered geology with thrust faults. Smoother than Salt.
Reference: Aminzadeh, Brac, and Kunz (1997).

Original (25 m): <https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Overthrust/fvp>

Resampled (50 m, 94x401x401): <https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Overthrust/fvp.r50>

2D section (slice i3=455, 187x801): <https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Overthrust/overthrust2D>

Reference wavefield (CBS, 50 m grid): https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Overthrust/BENCH_27PT/v.bin https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Overthrust/BENCH_27PT/cbs_overthrust_real.bin https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Overthrust/BENCH_27PT/cbs_overthrust_imag.bin

Grid: 2301 x 751 (n1=751 depth, n2=2301 lateral)
Spacing: 4 m
Notes: Classic 2D migration benchmark (Versteeg, 1994).

Velocity (vp.bin): <https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Marmousi/vp.bin>

Density (rho.bin): <https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Marmousi/rho.bin>

Grid: 1911 x 12601
Spacing: 6.25 m
Notes: Challenging sub-salt imaging (Billette and Brandsberg-Dahl, 2004).
Available from SEG open data.

SEG wiki page (registration may be required): https://wiki.seg.org/wiki/2004_BP_Velocity_Estimation_Benchmark_Model

Grid: 641 x 209 (n1=209 depth, n2=641 lateral)
Spacing: 25 m
Type: Acoustic VTI (vertical transverse isotropy)
Fields: Vp, epsilon, delta, eta, Vnmo, density
Notes: Representative of North Sea environments.
Exercises anisotropic operator (pure terms with
modified coefficients; cross-terms vanish for VTI).

Download (Geoazur WIND): https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Valhall2D/vp_true.bin https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Valhall2D/epsilon_true.bin https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Valhall2D/delta_true.bin https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Valhall2D/eta_true.bin https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Valhall2D/rho_true.bin

Grid: ~1911 x 12480
Spacing: 6.25 m
Type: Acoustic TTI (tilted transverse isotropy)
Fields: Vp, epsilon, delta, theta (tilt angle)
Notes: Canonical TTI benchmark (Shafiq et al., 2007).
Exercises full cross-derivative kernel with non-zero
tilt angles. This is the standard reference for TTI
stencil performance.

SEG wiki page (registration required): https://wiki.seg.org/wiki/2007_BP_Anisotropic_Velocity_Benchmark_Model

Synthetic 3D TTI (from Overthrust) No public 3D TTI model exists. The standard practice in GPU stencil papers is to overlay synthetic Thomsen parameters on the 3D Overthrust velocity model:

Vp: from Overthrust (801x801x187, 25 m)
epsilon: constant 0.1 - 0.2 (or linear gradient)
delta: constant 0.05 - 0.1
theta: smooth tilt (e.g., 20-45 degrees, linearly varying with depth)
phi: azimuthal angle (0 or smooth variation)

This gives a realistic velocity structure with controlled anisotropy parameters for benchmarking the TTI kernel (-d 9 mode).

Data hosting (Geoazur WIND project) All models above (except BP 2004) are hosted by the WIND project at Universite Cote d'Azur / Geoazur. Index page:

<https://www.geoazur.fr/WIND/bin/view/Main/Data/>

Usage with this benchmark Download the velocity binary and run:

```
wget https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Salt/fvp
./stencil.x -seg-salt -v fvp -t 1000
```

```
wget https://www.geoazur.fr/WIND/pub/nfs/FWI-DATA/GEOMODELS/Overthrust/fvp
./stencil.x -overthrust -v fvp -t 1000
```

The binary format is flat float32 with n1 (depth) as the fastest dimension. Velocities are in m/s; the driver squares them internally. Byte order is native (little-endian on x86).

Performance Metric

The standard metric for stencil codes is GPoints/s:

$$\text{GPoints/s} = (\text{nx} \times \text{ny} \times \text{nz} \times \text{nsteps}) / (\text{time} \times 1\text{e9})$$

This measures throughput independent of the internal algorithm (direct FD, GEMM-based, FFT, etc.) and is directly comparable across:

- Devito (Imperial College) -- symbolic PDE -> optimized C/OpenCL
- minimod (TotalEnergies) -- open-source GPU stencil mini-app
- Published results on PVC, A770, H100, MI300X

For reference, a memory-bandwidth-bound 8th-order stencil on PVC (HBM bandwidth ~3.2 TB/s) achieves approximately 200-400 GPoints/s depending on grid size and occupancy.

TTI (Anisotropic) Mode

With -d 9, the kernel computes the full tilted transverse isotropy operator with cross-derivative terms. Each cross-term uses SLM to buffer the intermediate result between two GEMM phases:

```
T = D_j x P          (first GEMM)
T = c_ij . T         (point-wise anisotropy scaling)
Y += D_i x T         (second GEMM)
```

SLM budget per cross-term: $2 \times K_PAD \times XMX_N \times \text{sizeof}(\text{ushort})$, rounded up to the DPAS K block size. Total for 3 cross-terms with barrier: fits within 128 KB (Xe2/BMG).

Operator Paths

The standard direct kernel applies one wide-reach operator (radius-4, 9-point) per dimension. The compact variants are independent runtime paths: they compile the same Ozaki-split BF16 x BF16 DPAS primitive with a smaller gather radius and rely on time evolution to build longer effective reach.

Methods (selected via STENCIL_METHOD environment variable):

0 = direct K=1, r=4 standard high-order (default) 1 = compact-r1 K=4, r=1 compact tridiagonal runtime path 2 = compact-r2 K=2, r=2 compact pentadiagonal runtime path 3 = compact-fit K=4, r=1 placeholder for dispersion-fitted coefficients

The “compact-fit” mode uses the free degrees of freedom in the K-factor coefficient product to match or exceed the dispersion quality of the standard 8th-order stencil at target frequencies, while retaining the memory savings of the cascade. Coefficients are precomputed once at initialization for the grid’s frequency content.

The first compact path is implemented for the isotropic apply kernel in methods 1-3. TTI cross-terms still use the direct two-phase DPAS path.

Environment variables controlling kernel specialization:

```
STENCIL_BF16      BF16-DPAS kernel (1=native BF16, 2=FP32-split via BF16)
STENCIL_BF16S     BF16 split wavefield storage format (0/1)
STENCIL_INT8      INT8-DPAS kernel (1=native INT8, 2=FP32-split via INT8)
STENCIL_METHOD    operator method (0-3, default 0)
STENCIL_STRIPS_PER_WG
    adjacent N-strips handled by one work-group (default: 2)
STENCIL_SG        subgroup size override (default: device preferred)
STENCIL_GRF256    force 256-GRF mode (0/1, default: auto)
STENCIL_TRIM      drop least-significant digit products (BF16 path)
STENCIL_LU        loop unroll strategy (-1=none, 0=inner, 1=outer)
STENCIL_LAYOUT    memory layout (0=XYZ, 1=blocked, 2=ZYX)
STENCIL_HALO      halo/padding size per axis
STENCIL_PML       enable PML absorbing boundary (0/1)
STENCIL_FP32_WG_X, STENCIL_FP32_WG_Y
    FP32 direct-kernel work-group shape (default: 32x8)
STENCIL_FP32_SBLOCK
    FP32 direct-kernel slow-axis SLM microblock (1 or 2,
    default: 2 except ZYX+PML)
```

STENCIL_FP32_BLOCK_IO

enable FP32 Intel 2D block reads for padded 32x8 cases (0/1, default: 0)

Specialized kernels are compiled on first use and cached in a thread-safe registry keyed by the method, compact-step parameters, strip grouping, subgroup size, packed specialization flags, FP32 work-group shape, grid shape, and term count. Subsequent launches with the same parameters are zero-cost.

Future extension: per-block adaptive method selection (different K in regions with different velocity/frequency content). This requires per-block dispatch logic and is not yet implemented.

Integration

The stencil API (`stencil_opencl.h`) accepts device pointers for all buffer arguments (wavefield, output, velocity). No host-device transfers happen inside the dispatch path – data stays on-device across the full time-stepping loop.

Initialization and USM control LIBXSTREAM provides `libxstream_init_config` for explicit control over the memory model before initialization:

```
libxstream_init_config_t cfg;
libxstream_init_config_default(&cfg); /* all fields = -1 (default) */
cfg.usm = 1; /* force Intel USM extensions */
cfg.device = 0; /* select device index */
libxstream_init_config(&cfg);
```

USM levels (`cfg.usm`):

```
-1 env/default (LIBXSTREAM_USM, or SVM coarse-grain fallback)
0 disable USM, force clCreateBuffer path
1 Intel USM extensions (clDeviceMemAllocINTEL)
2 OpenCL 2.0 SVM coarse-grain only
3 OpenCL 2.0 SVM with device-reported capabilities
```

When USM is active (level 1), device pointers from any source (SYCL, Level Zero, raw `clDeviceMemAllocINTEL`) can be passed to the stencil API without conversion.

SYCL interoperability SYCL USM device allocations (`sycl::malloc_device`) can be passed directly to `stencil_apply_laplacian` on Intel GPUs. The underlying mechanism is `clSetKernelArgMemPointerINTEL`, which accepts the same raw pointers that SYCL produces via the Level Zero unified runtime.

Requirements:

- Initialize LIBXSTREAM with `cfg.usm = 1`, or set the environment variable `LIBXSTREAM_USM=1`.
- The SYCL queue and the `libxstream` OpenCL context must target the same physical device (shared Level Zero context).
- Synchronization: use `sycl::queue::ext_oneapi_submit_barrier()` before calling `stencil_apply_laplacian`, and `libxstream_stream_sync` after, to order work correctly.
- No format conversion needed: float32 USM buffers are used as-is (velocity is expected as v^2).

External OpenCL consumers Any OpenCL application can call the API by passing `cl_mem`-backed device pointers obtained via `libxstream_mem_allocate`, or USM pointers from `clDeviceMemAllocINTEL` directly. The dispatch layer auto-detects USM availability and selects the appropriate kernel argument-setting path.

Scalar proxy (non-DPAS devices) The kernels include a scalar fallback that runs on any OpenCL 1.2+ device without DPAS/XMX hardware. This enables functional testing and correctness verification on iGPUs, CPUs, and other devices. Performance is not representative – the proxy exists purely for development and validation.

File Layout

```
samples/stencil/
  Makefile          build rules
  README.md         this file
  stencil.c         host driver (benchmark, model loading)
  stencil.py        Python benchmark harness (CSV/plot output)
  stencil_opengl.c  OpenGL context, kernel dispatch, exp_buf management
  stencil_opengl.h  public API, compile-time parameters
  stencil_kernels.h generated at build time from .cl sources
  kernels/
    stencil_common.cl parameters, gather macros, layout indexing
    stencil_fp32.cl   FP32 path: stencil_apply_direct (default)
    stencil_bf16.cl  BF16 path: stencil_apply, stencil_apply_tti
    stencil_int8.cl  INT8 path: stencil_apply_int8

libxstream/opencv/
  libxstream_common.h IEEE utilities, BF16 conversion, EXP2I, unroll macros
  libxstream_ozaki.h  Dekker split, BF16 DPAS macros
```