ParVec: Vectorized PARSEC Benchmarks

The software is provided "as is". We assume the users have some basic knowledge of how the PARSEC benchmark suite managing scripts work. For further info check the PARSEC documentation (http://parsec.cs.princeton.edu/).

Quick start guide

The first step is to point the managing script to the correct compiler/linker paths. Edit the "config/gcc.bldconf" file and:

1. Modify (if needed) the CC_HOME variable to point to your C/C++ compiler.
2. Modify (if needed) the MAKE variable to point to your make binary (set the number of threads, e.g., -j4).

Running "bin/parsecmgmt -a info" will provide a list of available benchmarks and configurations. SIMD versions are named parsec.simd.BENCHMARK_NAME. The "bin/" folder contains several examples for cleaning (simd_clean_all), building (simd_build_all_{x86, arm}) and running (simd_run_all_{x86, arm}). The scripts include Scalar and SIMD codes (SSE, AVX and NEON), both with and without the "hooks" API (check PARSEC documentation for details about "hooks"). The suite has been tested on Intel i7-2600K, i7-3770, i7-4770K processors using GCC 4.7.3 and 4.8.2 and Samsung Exynos 5410 using GCC 4.7.3.

List of additions

1. New configuration files ("config") for SSE, AVX and NEON, both with and without hooks.
2. New configuration files ("config/packages") for the different SIMD benchmarks.
3. New bibliography file ("config/bibliography/cebrian14parvec.bibconf") citing our ISPASS 2014 paper.
4. New per-benchmark configuration files for SSE, AVX and NEON. (/pkgs/{apps_simd/kernels_simd}/${benchmark_name}/parsec/)
5. SIMD wrapper and math libraries ("pkgs/libs/simd_libs"). Math library is based on the work from Julien Pommier, extended by Hallgeir Lien to work with NEON and AVX and then by Juan M. Cebrian for DP.

List of changes

This is a short list of changes to the different applications. No implementation details are provided.

1. Benchmarks now include "__parsec_thread_begin" and "__parsec_thread_end" calls that can be captured by the "hooks" API.
2. Updated YASM to version 1.2.0 (/pkgs/tools/yasm). Needed for the updated version of x264 (0.135.2345) .
3. All benchmarks
    1. New function call when threads star running and when threads finish. This is usefull to capture performance counter information on real systems.
4. Blackscholes
    1. Direct translation of the code to SIMD. No algorithmic changes.
5. Canneal
    1. New implementation using blocking/clustering (required and enabled by default in "clustering.h").
    2. Two functions to get the number of acceptable moves, one aims at speed, the other for precision. Since by default PARSEC input ends after a certain number of iterations, the fast function is enabled by default.
    3. The number of iterations in the clustered version is reduced. For block sizes of 64-256 items ((CLUSTER_ITEMS/4)*(CLUSTER_ITEMS/4)) achieves a better quality solution than the original non-blocked/clustered version.
    4. New atomic.h headers for ARM platforms included (from FreeBSD).
6. Fluidanimate
    1. Replaced data structures Array of Structures (AoS) to Structure of Arrays (SoA).
    2. Reverse loop order of ComputeDensities and ComputeForces to improve data reuse on the SIMD registers.
7. Raytrace (please read the comments of the author on this one)
    1. Fix a memory corruption problem when loading the data from file on SIMD mode (RTBOX).
8. Streamcluster
    1. SIMD version of the euclidean distance computation. Direct code translation. No algorithmic changes.
9. Swaptions
    1. Change order of j and b (iN and BLOCKSIZE to access randZ sequentially and be able to use SIMD).
    2. Direct translation of the code to SIMD, SIMD potential depends on BLOCKSIZE.
    3. Random number generator not vectorized, since the seed of each iteration is the previous generated number. An alternative would be to generate SIMD_WIDTH values and use them as seeds every iteration.

10. Vips
    1. SIMD versions of the convolution, linear transformation and interpolation filters.
    2. Convolution and interpolation filters would not get any benefit from AVX without completely changing the internal data representation. Due to the complexity of the application no AVX code is included.
11. x264
    1. Updated to version 0.135.2345, that supports both AVX and NEON.

Comments of the Author

While the wrapper library enables a common implementation for different target architectures (SSE, AVX, NEON), some benchmarks, like Vips and Fluidanimate have specific function calls for SSE, AVX or NEON that have not been covered by the wrapper library. This means that if a new instruction set is added to the wrapper library (e.g., AVX512) it will not work automatically for these benchmarks. Swaptions will also require slight modifications to include larger than 8 item SIMD registers due to the way the original implementation initializes the variables.
Complex benchmarks include the DEBUG_SIMD flag for verification purposes. In future versions we are considering replacing the math function headers with a complete match library (e.g., Yeppp, SLEEF, etc).

Blackscholes

We use custom exponential and logarithmic SIMD math functions. These functions have a different implementation from glibc/math. This produces slight variations on the benchmark outputs.

Canneal

The Canneal kernel is heavily memory bound. Our SSE implementation showed up no speedup, so there is no AVX or NEON implementation for the original code. However, a simple change to increase data locality (clustering/blocking) improves the performance of the application and enables a better SIMD implementation. This version is enabled by default in our distribution and exploited by the SIMD implementation.

Fluidanimate

Fluidanimate vectorization proved to be a challenge since it has many divergent branches, very limited number of elements per cell (at least for the native input in PARSEC) and other SIMD unfriendly characteristics. In addition, the verification for correctness of Fluidanimate output is very limited. Even the parallel version provides a complete different output from between runs (Fluidanimate is very susceptible to rounding errors on FP registers). This also affects the SIMD implementation. For verification enable the DEBUG_SIMD flag in the code, it will show up the maximum difference between the output obtained using the scalar code and the SIMD code.

Raytrace

The Raytrace benchmark provided in PARSEC 3.0b is written using SSE. The scalar version is obtained from a wrapper library that translates SSE code into “for” loops. Even though it provides a “valid” output, we believe the conclusions obtained from running the scalar Raytrace would be those of running a “SSE emulator”, not a real raytracing application (the fact that the SSE version runs 25+ faster than the scalar version proves there is something wrong with it). We did not include any AVX or NEON version since we believe the benchmark should be replaced.

Streamcluster

NEON version runs slower than the scalar code. We believe that the memory subsystem on the ARM development boards we have tested (Exynos 5250/5410) cannot keep up with the memory requirements on the NEON code.

Swaptions

The speedup of this application is limited by the block size of the blocking/clustering implementation and the inherent data dependencies of the algorithm. A SIMD version of the random number generator could further improve the speedup of the application and match that of commercial libraries (LIBOR).

Vips

Recent versions of Vips (after 7.24) include support for SIMD using “Orc” for convolution, erode, dilate and add. However, after checking this newer version we saw no major difference when using PARSEC input and running in benchmark mode. Due to the complexity of the application we decided to focus our efforts in the most time consuming filters of the benchmark mode (convolution, linear transformation and interpolation).