# A SSE-Based Implementation of a Ray Tracer for the Comparision with Coprocessors in Hybrid Computing System

Volker Hampel and Erik Maehle

Insititute of Computer Engineering
University of Lübeck
Ratzeburger Allee 160
23562 Lübeck, Germany
hampel, maehle@iti.uni-luebeck.de

**Abstract:** In this paper we present and discuss our efforts to accelerate a sample application by using the Streaming SIMD Extensions (SSE) to the x64 instruction set. Several approaches to their integration into the source code are tested and evaluated against each other. They are assembler intrinsics, the initial source code combined with different compiler flags, and enhanced code for better SSE inference. Their performances are compared to benchmarks from two hybrid computing systems, which use a Field Programmable Gate Array (FPGA) and a Graphics Processing Unit (GPU), respectively. As the interfaces to manipulated/accelerated code sections are the same in all cases, comparability always is maintained.

## 1 Motivation

Hybrid computing, the combination of a standard Central Processing Unit (CPU) with devices like Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs), aims at utilizing these special architectures to accelerate applications. It profits from massively parallel floating-point execution units in GPUs and from the flexibility and bit-level parallelism in FPGAs. In most cases these devices serve as a coprocessor which is added to a computer system. Being a separate device, the CPU and the coprocessor do not share the memory and communication usually is handled explicitly from within an application. Both of these aspects add to a bottleneck when using an application specific coprocessor. Results from earlier work [6] have shown, that although this bottleneck has been carefully taken into account when developing a hybrid application, only moderate speedups are achieved. So, what if the hybrid approach was dropped in favor of a tweaked one, which aims at using the special architectural features of current CPUs? One such architectural feature are the Streaming SIMD Extensions (SSE), which we use to accelerate a sample application in order to compare its performance with the hybrid implementations' performances. Several ways of using the SSE are tested and evaluated against each other.

First, background information is given, comprising more details on the SSE and a sample application. Details on the different ways to use SSE are presented in the following section. They are evaluated subsequently by presenting the method of measurement for comparability and the actual results. We conclude this article by discussing the results against architectural aspects of the evaluation systems and an outlook aimed at future architectural features.

## 2 Background

This section gives brief overviews of the SSE instruction set extensions and a ray tracer which is used as a sample application.

### 2.1 Streaming SIMD Extensions (SSE)

The Streaming SIMD Extensions (SSE) add vector processing capacity to common, scalar CPUs. Scalar processors execute a single instruction on a set of single operands, which maps to the Single Instruction stream Single Data stream (SISD) category of Flynn's taxonomy [4]. Vector processing executes a single instruction on a set of multiple sets of operands, accordingly mapping to the Single Instruction stream Multiple Data streams (SIMD) category of Flynn's taxonomy. The benefit of instructions of this category is an increased parallelism during execution at the cost of additional hardware resources.

With SSE3 several groups of instructions are available: The *packed* group comprises the typical vector instructions as described above. In addition, *scalar* instructions do not utilize multiple data streams but are also executed on SSE's special register set, preventing to move data back and forth to the standard register set. Besides these two main groups, *horizontal* instructions use multiple components of a single vector as operands.

The vectors in SSE3 are 128 bit long, allowing to be split into 16, 8, 4, or 2 values depending in the values' types. Concerning floating point numbers, both single and double precision is supported, resulting in vectors of four and two components, respectively. Detailed information on the available instructions can be found in [9, 10, 11].

### 2.2 Sample Application

A simple ray tracer has been implemented to serve as a standardized sample application in all performance evaluations. It is based on [14]. The rendering algorithm has been ported from C++ to pure C while maintaining a C++-based windowing environment to control the ray tracer and to visualize the rendering progress. As this work is not about optimizing ray tracing itself, only few effects have been implemented, keeping the physical model quite simple. However, the computational load is large enough to detect accelerations.

Ray tracing is the simulation of geometric optics, meaning that a ray is sent starting from a picture plane. The ray intersects with objects in a scene, and following the objects' surface properties the ray is colorized and reflected rays may be generated and traced, simulating reflections. Here, only three types of objects, planes, spheres, and boxes, are supported as each type requires a different formula to evaluate ray intersection. In addition to the color of the objects, light sources are simulated to add shadows to the scene and picture. For all points, in which a ray intersects with an object, a shadow ray from this point to each light source is generated and traced. If such a shadow ray intersects with an object, its starting point is not lighted by the rays destination, the light source. Consequently, its color is darkened compared with the opposite case. Within our implementation, the light sources are one statistical distributed ambient light and a variable number of point lights.

As the class structure of the initial implementation of the rendering algorithm has been abandoned, the properties of the objects, lights, surfaces, and rays are kept in memory sections as arrays of structures. The rendering algorithm itself runs through a series of independent steps which access the different memory sections. In addition, all these steps are not performed on a single ray but a set of rays. Both these measures better support memory mapped coprocessors as it is (a) easier to select parts of the functionality to bring to the coprocessor and it is (b) easier to establish an efficient, bulk-based communication with the device [6].

The FPGA-based hybrid evaluation system provides only a limited buffer capacity for the memory sections, which are accessed by the respective functionality. Therefore, the whole picture is split into tiles, and the tiles are injected to the rendering algorithm.

## 3 Implementations

In this section different implementations using the instruction set extensions are presented. All of them focus on the evaluation of the functions to find out if a shadow ray intersects with an object on its way to a light source. The different implementations make use of assembler intrinsics, enhanced code which aims at a better inference of SSE-instructions, and the compiler and its flags.

### 3.1 Assembler Intrinsics

Using assembler code is aimed at taking over full control of how a calculation is executed on the CPU. It requires the programmer to carefully analyze the function to be implemented and to plan and to schedule its execution. Here, a graphical approach for this analysis and the scheduling is exerted: A Data Flow Graph (DFG) of the function is derived from the initial C code, visualizing the operations and their dependencies. All the input, output, and intermediate data are packed and assigned to SSE-registers to allow parallel instructions to be executed on them. Figure 1 shows the DFG for this function for planes. The graph assigning registers and instructions is shown in Fig. 2.

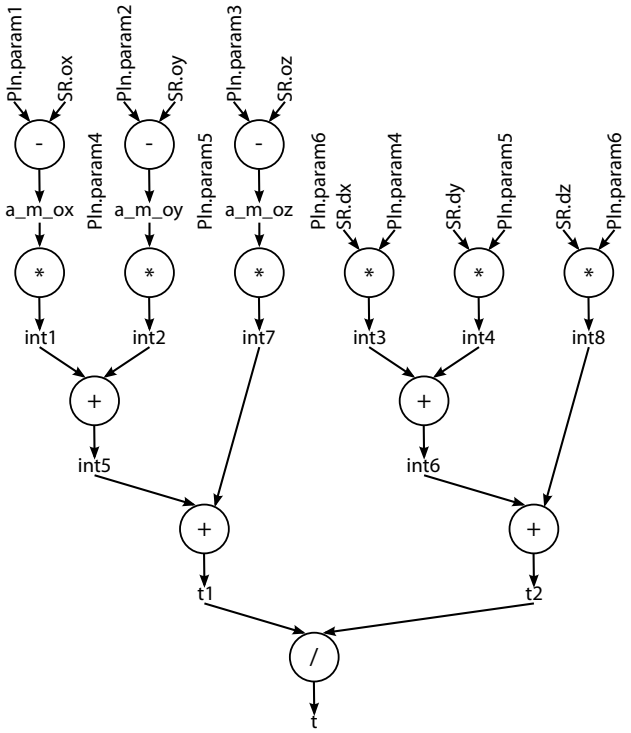


Figure 1: Data Flow Graph (DFG) showing the dependencies among the operations and from the input data for a plane's intersection function for shadow rays.

The properties of a plane (`Pln`) and the shadow rays (`SR`) are available under consecutive addresses in the system memory, allowing to load two double precision values into a register with a single instruction. A series of packed double vector instructions is then used to calculate intermediate results `int5` and `int6`. A three-dimensional physical model is used for ray tracing in which all vectors and points have three double precision components and coordinates, respectively. The packed instructions used so far were able to process two of these components and coordinates. The third ones are now loaded to registers with scalar instructions. A scalar input value (`SR.dz`) and an intermediate result (`a_m_dz`) can then be combined into a register to allow using packed and horizontal double precision instructions to give the final results `t1` and `t2`.

Comparing the graphs in Fig. 1 and Fig. 2 with each other shows that the number of arithmetic instructions has been reduced from 14 to 7. To evaluate the functions another calculation is executed, which gives the distance between the starting point of the shadow ray and the light source. For this calculation the number of instructions can be reduced from 9 to 7. In total, 14 SIMD instructions replace 23 SISD instructions, thus giving an upper bound for a possible speedup of 1.64.

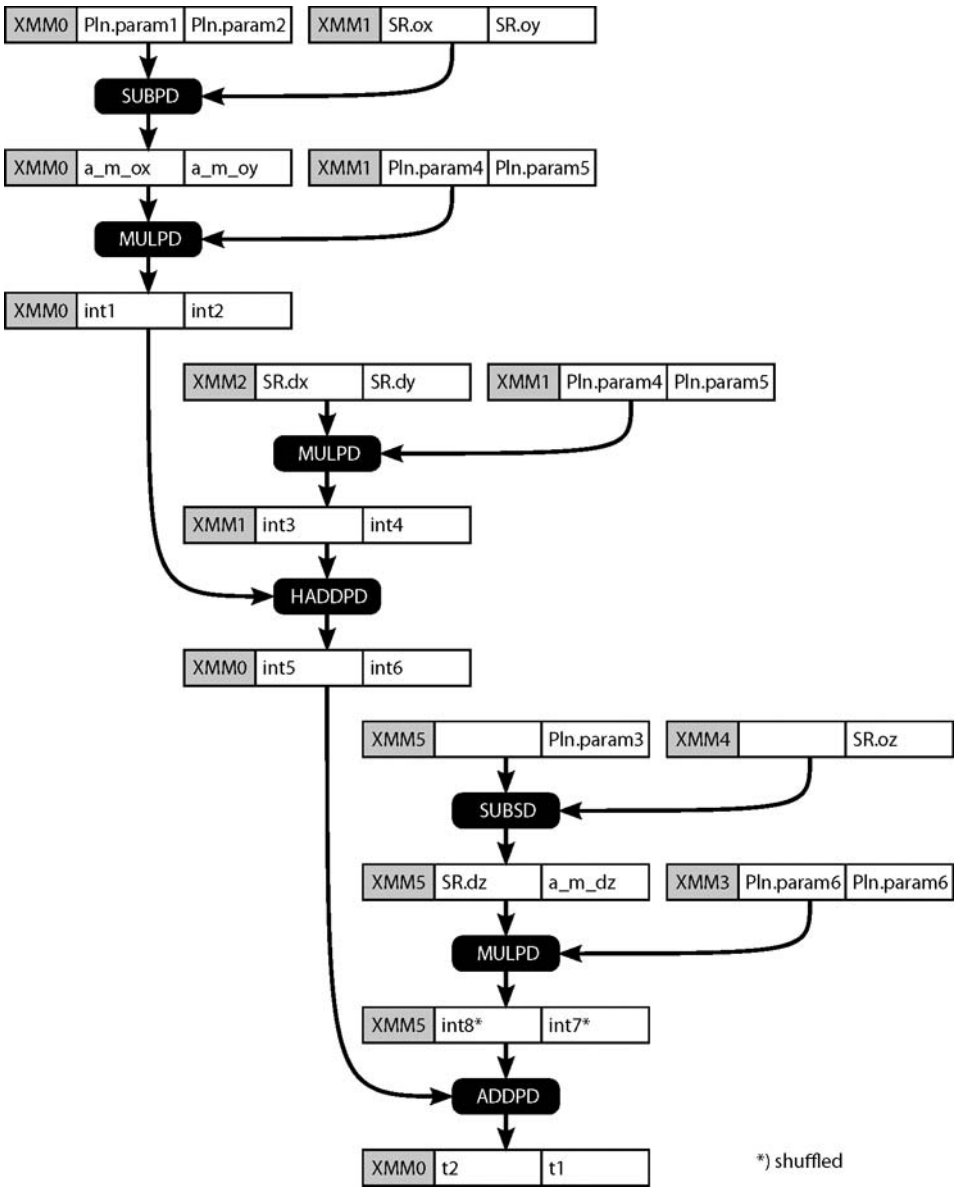


Figure 2: Scheduled DFG assigning data to registers and registers to instructions.

### 3.2 Enhanced Code

While staying in the domain of pure C-code, certain coding techniques are used to parallelize the execution of some calculations. With the planes and the spheres, iterations always cover two shadow rays which are mapped to the upper and the lower part of the SSE-registers. Doing so should allow to keep the sequence of calculations but working on twice the data. With boxes, calculations are more dependent on intermediate results, leading to a very fine grained computational load. Hence, only one shadow ray is iterated at a time and parallelism is exploited on a local basis.

To parallelize the calculation vector types have to be defined. Following [13] the size of the vector in bytes has to be set, which is 16 for the 128 bit wide SSE-registers. Defining a `union` allows to access a variable of the vector type as a whole or by its components:

```
typedef double v2df __attribute__ ((vector_size(16)));
union twodoubles {
   double s[2];
   v2df v;
};
```

Within the actual code, instances of this union of two double floating point numbers are declared. Vector operations can now be coded by referencing the vector in the unions' instances. The operators are overloaded to directly accept vectors as operands, and the compiler maps this code to SSE-instructions:

```
union twodoubles a_minus_o_x;
a_minus_o_x.v = pln_tmp_v_p1.v - sdr_tmp_v_ox.v;
```

### 3.3 The Compiler

The main reason to have a closer look at the compiler and its configuration is to generate a reference implementation that does not utilize the SSE instructions, i. e., to use the standard 387-Floating Point Unit (FPU) in x64-CPUs. The flag `-mfpmath=387` can be used to choose the said FPU. In addition to this reference implementation, the compiler is also instructed to use the SSE-FPU and, as a GCC extra, to try to use both FPU-units if there are two distinct ones. In addition to manually select the FPU, all implementations are compiled without any optimization effort `-o0` and with a high optimization effort `-o3`.

## 4 Evaluation

This paper's work is aimed to generate comparative performances of a ray tracer, which is based on modern CPUs only. Therefore we first describe how comparability is maintained over all implementations and systems. The two evaluations systems used in all our performance studies are briefly described before presenting the actual results.

### 4.1 Maintaining Comparability

In Sec. 1 it has been mentioned that for the implementations of the ray tracer based on a hybrid computing system, that those parts to be brought to the coprocessor have been carefully chosen to have a good trade-off between computational load and the communication effort. The measure for this trade-off is arithmetic intensity [7], which is systematically evaluated for the whole rendering algorithm in [6]. The results from this work suggest that the evaluation of the intersection functions of the different object types for shadow rays should be implemented on the coprocessor. The reasons are a high computational load, modest communication effort, and diverging the execution path to the coprocessor twice for the ambient light and the point lights, without any reconfiguation.

To measure the performances two time-stamps are taken, the first just before calling the shadow hit functions and the second just after returning from them. The difference of both stamps gives the execution time, no matter where or how the calculations are done. Outside these time-stamps everything must remain unchanged, forcing all implementations to use and also maintain the data structures of the whole ray tracer. By doing so, the impact of the different architectural approaches can directly be compared with each other.

### 4.2 Evaluation Systems

The evaluation system is one node of a Cray XD1 cluster computer [2]. It comprises two dual-core AMD Opteron 275 processors running at 2.2 GHz, which implement the x64 instruction set plus SSE3 extensions. A node's four cores and 4 GBytes of memory build a Symmetric Multiprocessor (SMP). Only one of the cores is used by the rendering algorithm. The Cray XD1 node also hosts a Xilinx Virtex4 LX160 FPGA, which is mapped into the node's memory address space. Technologically the FPGA connects to the SMP over a reduced implementation of HyperTransport, which provides two bidirectional communication channels. Depending on the clock speed of the FPGA each of the two channels' max. bandwidth is 1.6 GBytes/s. The actual coprocessor runs at 149 MHz.

Unfortunately, a GPU cannot be integrated into a Cray XD1 node. It is instead hosted in a commodity PC over PCIe 2.0 x16. The GPU is a Nvidia GTX 285 [12], the host system is a Intel Core 2 Quad Q8200 Yorkfield-6M CPU running at 2.33 GHz.

### 4.3 Results

Table 1 gives the runtimes of the evaluation of the shadow hit functions of all the objects for ambient light (ao), two point lights (pl), and the whole rendering (all). The times are given for all five possibilities introduced in Sec. 3 with and without compiler optimizations. For each implementation the first line gives a mean value of the runtime, the second line compares the implementation's performance to the fastest one within a group of light and optimization. The third line gives the speedup through compiler optimization.

Table 1: Performances of the implementations from Sec. 3 of the evaluation of the shadow hit functions. Runtimes in microseconds.

| | `-o0` | | | `-o3` | | | |
|---|---|---|---|---|---|---|---|
| | **ao** | **pl** | **all** | **ao** | **pl** | **all** | **speedup...** |
| | 6 201 | 15 113 | 54 868 | 6 206 | 15 130 | 55 176 | |
| **ASM** | 1.022 | 1.000 | 1.000 | 1.791 | 1.973 | 1.568 | relative to ranking |
| | | | | 0.999 | 0.999 | 0.994 | through opt. |
| | 6 069 | 17 477 | 57 553 | 3 465 | 7 667 | 35 194 | |
| **SSE** | 1.000 | 1.156 | 1.049 | 1.000 | 1.000 | 1.000 | relative to ranking |
| | | | | 1.751 | 2.280 | 1.635 | through opt. |
| | 6 140 | 17 822 | 56 915 | 4 057 | 8 973 | 38 357 | |
| **387** | 1.012 | 1.179 | 1.037 | 1.171 | 1.170 | 1.090 | relative to ranking |
| | | | | 1.513 | 1.986 | 1.484 | through opt. |
| | 6 103 | 17 924 | 57 974 | 6 090 | 17 904 | 57 579 | |
| **both** | 1.006 | 1.186 | 1.057 | 1.758 | 2.335 | 1.636 | relative to ranking |
| | | | | 1.002 | 1.001 | 1.007 | through opt. |
| | 8 027 | 20 086 | 61 763 | 4 332 | 9 494 | 36 972 | |
| **enh.** | 1.323 | 1.329 | 1.126 | 1.250 | 1.238 | 1.051 | relative to ranking |
| | | | | 1.853 | 2.116 | 1.671 | through opt. |
| | 3 080 | 6 032 | 40 205 | 25 402 | 62 365 | 112 077 | |
| **FPGA** | 1.970 | 2.505 | 1.365 | 1.125 | 1.271 | 1.142 | vs. fastest from above |
| | | | | 0.123 | 0.097 | 0.359 | through opt. |
| **GPU SW** | 5 039 | 13 093 | 39 934 | 2 490 | 4 614 | 21 012 | |
| | 589 | 943 | 22 005 | 583 | 943 | 14 301 | |
| **GPU HW** | 11.689 | 13.884 | 1.815 | 4.271 | 4.890 | 1.469 | against GPU SW |
| | | | | 1.010 | 1.000 | 1.539 | through opt. |

The performance of the FPGA-based ray tracer is also given in Tab. 1, and it is directly compared to the best performing implementation within the respective light/optimization-group. Because the compiler optimizations cause the FPGA's performances to plummet, the better `-o0`-performances are compared to the best `-o3`-values of the five software based implementations. Finally, the runtimes and speedups of the GPU-based ray tracer are given based on the second evaluation system.

## 5 Discussion

Without compiler optimizations either FPU performs equally well, and there is no benefit from telling the compiler to try to use both. According to [3] all instructions are split into

microoperations which are then issued to two 64 bit wide FPUs. However, only one of the FPUs can execute floating point multiplications, while the other performs additions. Consequently, the compiler must fail when trying to use both 387- and SSE-FPUs as all floating point instructions are mapped to the same resource. Because of the width of the FPUs, packed double SSE-instructions can not be executed in parallel as assumed. Instead they are injected into the floating point pipeline in two clock cycles. When the compiler optimizes the code, it fares better when aiming at SSE-instructions instead of 387 ones. A reason might be a better performance with packed memory accesses. The compiler achieves no speedup with optimizations when trying to use the (fictional) two SSE- and 387-FPUs.

The assembler intrinsics lead to a light speedup for the point lights over the next-ranking SSE-based implementation, while computations for the ambient lighting are almost at par. The compiler fails to optimize the code based on intrinsics, as they are supposed to bypass the compiler, and they consequently remain untouched.

The enhanced code totally fails to bring any speedup, as it performs 24-25% and 32-33% slower than the initial code with and without optimizations, respectively.

As Tab. 1 shows, no additional speedups are achieved by using vector extensions to the CPU. The compiler already makes good use of the processor, and actually no 128 bit wide processing units are available.

## 6 Outlook

In Sec. 4.2 the evaluation systems are introduced as quad core machines. Non-hybrid systems can thus achieve speedups by exploiting thread-based parallel programming on shared memory systems. Multithreading has been neglected so far, but promises an easy speedup of 2-3. With this additional speedup, the multithreaded CPU would outperform the FPGA-based hybrid system and it would catch up the GPU-based system. The expected performances would meet the findings in [8], where the two to three magnitudes speedups of especially GPU-based systems are challenged. They conclude that extraordinary speedups are the results of not using all the capabilities of modern CPUs and that speedups from 0.5 to 2 were more realistic.

Advanced Vector Extensions (AVX) have been implemented in the most recent processors, promising a further speedup by 2 as the vector length is doubled to 256 bit. Again technology blogs on Intel's Sandy Bridge Microarchitecture [5] reveal a total of three FPUs, of which only one is an adder and another is a multiplier. They are all 128 bit wide, allowing to process two double precision floating point numbers, instead of four as the vector length suggests. Future work should ensure that compilation makes use of these most recent instruction set extensions.

As AMD has introduced its first Fusion processors [1], which combine a graphics core with several CPU cores on a single chip, future work should focus on how much applications can benefit from such highly integrated hybrid machines. We expect the communication bottleneck to the graphics core to be largely overcome.

## References


[1] Brookwood, N.: AMD Fusion Family of APUs: Enabling a Superior, Immersive, PC Experience. AMD White Paper, March 2010

[2] Cray Inc.: Cray XD1 System Overview (S-2429-141). 2006.

[3] Doerfler, D., Hensinger, D., Leback, B., and Miles, D.: Tuning C++ Applications for the Latest Generation x64 Processors with PGI Compilers and Tools. Proceedings of the Cray User Group (CUG) 2007.

[4] Flynn, M. J.: Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers, vol. C-21, no. 9, pp. 948–960, 1972.

[5] Gavrichenkov, I.: Intel Sandy Bridge Microarchitecture Preview. 27 December 2010, xbitlabs.com, retrieved 10 May 2011. `http://www.xbitlabs.com/articles/cpu/display/sandy-bridge-microarchitecture.html`

[6] Hampel, V., Grigori, G. and Maehle, E.: A Code-based Analytical Approach for Using Separate Device Coprocessors in Computing Systems. Architecture of Computing Systems - ARCS 2011, LNCS, Berekovic, M., Fornaciari, W., Brinkschulte, U. und Silvano, C. (Hrsg.), S. 1–12, 2011, Springer Berlin/Heidelberg.

[7] Harris, M.: Mapping Computational Concepts to GPUs. GPU Gems 2, Chapter 31, edited by Mark Pharr. 2005, Addison-Wesley Longman, Amsterdam, NL.

[8] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P.: Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA), Saint-Malo, France, ACM, 2010.

[9] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, Juni 2010.

[10] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M, Juni 2010.

[11] Intel Corporation: Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, N-Z, Juni 2010.

[12] Nvidia Corporation: Technical Brief NVIDIA GeForce GTX 200 GPU Architectural Overview. May 2008.

[13] Stallman, R. M.: Using the GNU Compiler Collection. Free Software Foundation Inc., 2010.

[14] Suffern, K. G.: Ray Tracing from the Ground up. A K Peters Ltd., 2007.