(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
18 April 2024 (18.04.2024)

WIPO | PCT

(10) International Publication Number
WO 2024/079686 A1

(51) International Patent Classification:
G06F 8/41 (2018.01)

(21) International Application Number:
PCT/IB2023/060302

(22) International Filing Date:
12 October 2023 (12.10.2023)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
63/415,303 12 October 2022 (12.10.2022) US

(71) Applicant: **MOBILEYE VISION TECHNOLOGIES LTD.** [IL/IL]; 13 Hartom St., 9777513 Jerusalem (IL).

(72) Inventors: **MARJIEH, Michael**; 22 Safuriya, 1650622 Nazareth (IL). **KOM, Alon**; 8 Zohar Street, 3540234 Haifa (IL). **BENITA BEN-SIMHON, Oren**; 38 Abba Hushi Street, 3478117 Haifa (IL).

(74) Agent: **SHICHRUR, Naim Avraham**; SHICHRUR & CO., Horowitz Tower, Suite 521, 2 Moti Kind Street, 7638523 Rehovot (IL).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CV, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IQ, IR, IS, IT, JM, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, MG, MK, MN, MU, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA, SC, SD, SE, SG, SK, SL, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, WS, ZA, ZM, ZW.

(84) Designated States *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, CV, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SC, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ,

(54) Title: APPARATUS, SYSTEM, AND METHOD OF COMPILING CODE FOR A PROCESSOR

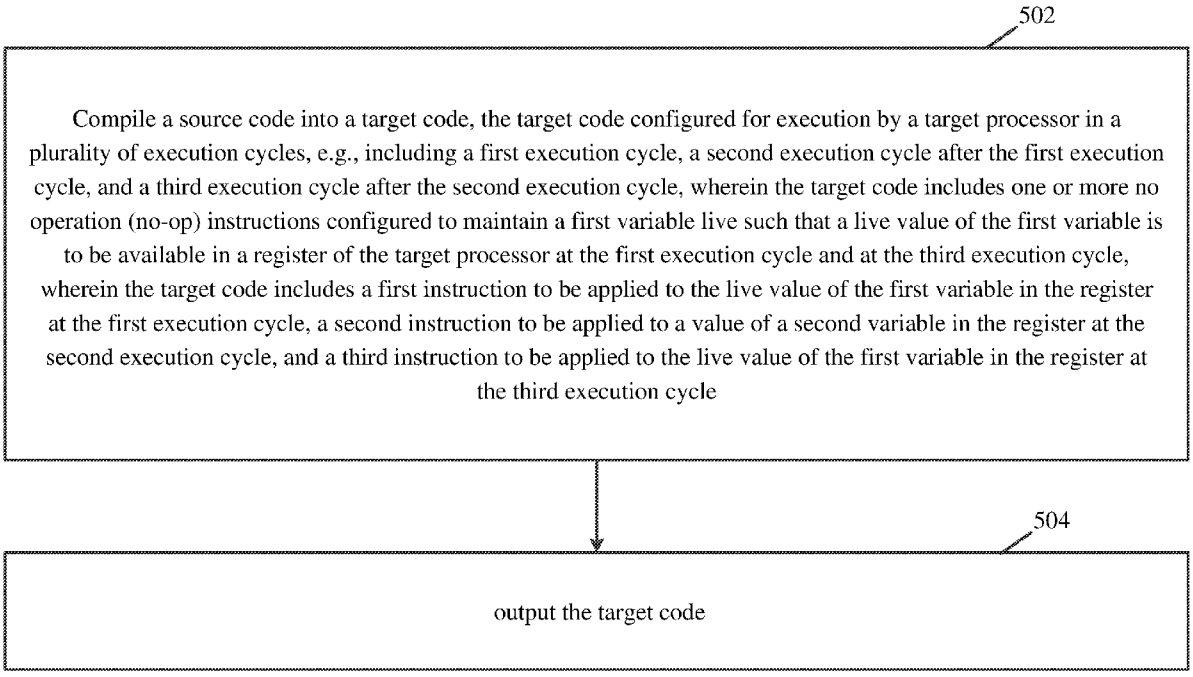

Fig. 5

(57) Abstract: For example, a compiler may be configured to compile a source code into a target code configured for execution by a target processor in a plurality of execution cycles including a first execution cycle, a second execution cycle after the first execution cycle, and a third execution cycle after the second execution cycle. For example, the target code may include one or more no operation (no-op) instructions configured to maintain a first variable live such that a value of the first variable is to be available in a register of the target processor at the first execution cycle and at the third execution cycle.

WO 2024/079686 A1

[Continued on next page]

DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, ME, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *as to applicant's entitlement to apply for and be granted a patent (Rule 4.17(ii))*
— *of inventorship (Rule 4.17(iv))*

Published:

— *with international search report (Art. 21(3))*
— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*
— *in black and white; the international application as filed contained color or greyscale and is available for download from PATENTSCOPE*

APPARATUS, SYSTEM, AND METHOD OF COMPILING CODE FOR A PROCESSOR

CROSS REFERENCE

[0001] This Application claims the benefit of and priority from US Provisional Patent Application No. 63/415,303 entitled "APPARATUS, SYSTEM, AND METHOD OF COMPILING CODE FOR A PROCESSOR", filed October 12, 2022, the entire disclosure of which is incorporated herein by reference.

BACKGROUND

[0002] A compiler may be configured to compile source code into target code configured for execution by a processor.

[0003] There is a need to provide a technical solution to support efficient processing functionalities.

BRIEF DESCRIPTION OF THE DRAWINGS

[0004] For simplicity and clarity of illustration, elements shown in the figures have not necessarily been drawn to scale. For example, the dimensions of some of the elements may be exaggerated relative to other elements for clarity of presentation. Furthermore, reference numerals may be repeated among the figures to indicate corresponding or analogous elements. The figures are listed below.

[0005] Fig. 1 is a schematic block diagram illustration of a system, in accordance with some demonstrative aspects.

[0006] Fig. 2 is a schematic illustration of a compiler, in accordance with some demonstrative aspects.

[0007] Fig, 3 is a schematic illustration of a vector processor, in accordance with some demonstrative aspects.

[0008] Fig. 4 is a schematic flow-chart illustration of a method of compiling code for a processor, in accordance with some demonstrative aspects.

[0009] Fig. 5 is a schematic flow-chart illustration of a method of compiling code for a processor, in accordance with some demonstrative aspects.

[00010] Fig. 6 is a schematic illustration of a product, in accordance with some demonstrative aspects.

DETAILED DESCRIPTION

[00011] In the following detailed description, numerous specific details are set forth in order to provide a thorough understanding of some aspects. However, it will be understood by persons of ordinary skill in the art that some aspects may be practiced without these specific details. In other instances, well-known methods, procedures, components, units and/or circuits have not been described in detail so as not to obscure the discussion.

[00012] Some portions of the following detailed description are presented in terms of algorithms and symbolic representations of operations on data bits or binary digital signals within a computer memory. These algorithmic descriptions and representations may be the techniques used by those skilled in the data processing arts to convey the substance of their work to others skilled in the art.

[00013] An algorithm is here, and generally, considered to be a self-consistent sequence of acts or operations leading to a desired result. These include physical manipulations of physical quantities. Usually, though not necessarily, these quantities capture the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers or the like. It should be understood, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities.

[00014] Discussions herein utilizing terms such as, for example, “processing”, “computing”, “calculating”, “determining”, “establishing”, “analyzing”, “checking”, or the like, may refer to operation(s) and/or process(es) of a computer, a computing platform, a computing system, or other electronic computing device, that manipulate and/or transform data represented as physical (e.g., electronic) quantities within the computer’s registers and/or memories into other data similarly represented as physical quantities within the computer’s registers and/or memories or other information storage medium that may store instructions to perform operations and/or processes.

[00015] The terms “plurality” and “a plurality”, as used herein, include, for example, “multiple” or “two or more”. For example, “a plurality of items” includes two or more

items.

[00016] References to “one aspect”, “an aspect”, “demonstrative aspect”, “various aspects” etc., indicate that the aspect(s) so described may include a particular feature, structure, or characteristic, but not every aspect necessarily includes the particular feature, structure, or characteristic. Further, repeated use of the phrase “in one aspect” does not necessarily refer to the same aspect, although it may.

[00017] As used herein, unless otherwise specified the use of the ordinal adjectives “first”, “second”, “third” etc., to describe a common object, merely indicate that different instances of like objects are being referred to, and are not intended to imply that the objects so described must be in a given sequence, either temporally, spatially, in ranking, or in any other manner.

[00018] Some aspects, for example, may capture the form of an entirely hardware aspect, an entirely software aspect, or an aspect including both hardware and software elements. Some aspects may be implemented in software, which includes but is not limited to firmware, resident software, microcode, or the like.

[00019] Furthermore, some aspects may capture the form of a computer program product accessible from a computer-usable or computer-readable medium providing program code for use by or in connection with a computer or any instruction execution system. For example, a computer-usable or computer-readable medium may be or may include any apparatus that can contain, store, communicate, propagate, or transport the program for use by or in connection with the instruction execution system, apparatus, or device.

[00020] In some demonstrative aspects, the medium may be an electronic, magnetic, optical, electromagnetic, infrared, or semiconductor system (or apparatus or device) or a propagation medium.

[00021] In some demonstrative aspects, a data processing system suitable for storing and/or executing program code may include at least one processor coupled directly or indirectly to memory elements, for example, through a system bus. The memory elements may include, for example, local memory employed during actual execution of the program code, bulk storage, and cache memories which may provide temporary storage of at least some program code in order to reduce the number of times code must

be retrieved from bulk storage during execution.

[00022] In some demonstrative aspects, input/output or I/O devices (including but not limited to keyboards, displays, pointing devices, etc.) may be coupled to the system either directly or through intervening I/O controllers. In some demonstrative aspects, network adapters may be coupled to the system to enable the data processing system to become coupled to other data processing systems or remote printers or storage devices, for example, through intervening private or public networks. In some demonstrative aspects, modems, cable modems and Ethernet cards are demonstrative examples of types of network adapters. Other suitable components may be used.

[00023] Some aspects may be used in conjunction with various devices and systems, for example, a computing device, a computer, a mobile computer, a non-mobile computer, a server computer, or the like.

[00024] As used herein, the term "circuitry" may refer to, be part of, or include, an Application Specific Integrated Circuit (ASIC), an integrated circuit, an electronic circuit, a processor (shared, dedicated or group), and/or memory (shared. Dedicated, or group), that execute one or more software or firmware programs, a combinational logic circuit, and/or other suitable hardware components that provide the described functionality. In some aspects, some functions associated with the circuitry may be implemented by, one or more software or firmware modules. In some aspects, circuitry may include logic, at least partially operable in hardware.

[00025] The term “logic” may refer, for example, to computing logic embedded in circuitry of a computing apparatus and/or computing logic stored in a memory of a computing apparatus. For example, the logic may be accessible by a processor of the computing apparatus to execute the computing logic to perform computing functions and/or operations. In one example, logic may be embedded in various types of memory and/or firmware, e.g., silicon blocks of various chips and/or processors. Logic may be included in, and/or implemented as part of, various circuitry, e.g., processor circuitry, control circuitry, and/or the like. In one example, logic may be embedded in volatile memory and/or non-volatile memory, including random access memory, read only memory, programmable memory, magnetic memory, flash memory, persistent memory, and the like. Logic may be executed by one or more processors using memory, e.g., registers, stuck, buffers, and/or the like, coupled to the one or more processors,

e.g., as necessary to execute the logic.

[00026] Reference is now made to Fig. 1, which schematically illustrates a block diagram of a system 100, in accordance with some demonstrative aspects.

[00027] As shown in Fig. 1, in some demonstrative aspects system 100 may include a computing device 102.

[00028] In some demonstrative aspects, device 102 may be implemented using suitable hardware components and/or software components, for example, processors, controllers, memory units, storage units, input units, output units, communication units, operating systems, applications, or the like.

[00029] In some demonstrative aspects, device 102 may include, for example, a computer, a mobile computing device, a non-mobile computing device, a laptop computer, a notebook computer, a tablet computer, a handheld computer, a Personal Computer (PC), or the like.

[00030] In some demonstrative aspects, device 102 may include, for example, one or more of a processor 191, an input unit 192, an output unit 193, a memory unit 194, and/or a storage unit 195. Device 102 may optionally include other suitable hardware components and/or software components. In some demonstrative aspects, some or all of the components of one or more of device 102 may be enclosed in a common housing or packaging, and may be interconnected or operably associated using one or more wired or wireless links. In other aspects, components of one or more of device 102 may be distributed among multiple or separate devices.

[00031] In some demonstrative aspects, processor 191 may include, for example, a Central Processing Unit (CPU), a Digital Signal Processor (DSP), one or more processor cores, a single-core processor, a dual-core processor, a multiple-core processor, a microprocessor, a host processor, a controller, a plurality of processors or controllers, a chip, a microchip, one or more circuits, circuitry, a logic unit, an Integrated Circuit (IC), an Application-Specific IC (ASIC), or any other suitable multi-purpose or specific processor or controller. Processor 191 may execute instructions, for example, of an Operating System (OS) of device 102 and/or of one or more suitable applications.

[00032] In some demonstrative aspects, input unit 192 may include, for example, a

keyboard, a keypad, a mouse, a touch-screen, a touch-pad, a track-ball, a stylus, a microphone, or other suitable pointing device or input device. Output unit 193 may include, for example, a monitor, a screen, a touch-screen, a flat panel display, a Light Emitting Diode (LED) display unit, a Liquid Crystal Display (LCD) display unit, a plasma display unit, one or more audio speakers or earphones, or other suitable output devices.

[00033] In some demonstrative aspects, memory unit 194 includes, for example, a Random Access Memory (RAM), a Read Only Memory (ROM), a Dynamic RAM (DRAM), a Synchronous DRAM (SD-RAM), a flash memory, a volatile memory, a non-volatile memory, a cache memory, a buffer, a short term memory unit, a long term memory unit, or other suitable memory units. Storage unit 195 may include, for example, a hard disk drive, a Solid State Drive (SSD), or other suitable removable or non-removable storage units. Memory unit 194 and/or storage unit 195, for example, may store data processed by device 102.

[00034] In some demonstrative aspects, device 102 may be configured to communicate with one or more other devices via at least one network 103, e.g., a wireless and/or wired network.

[00035] In some demonstrative aspects, network 103 may include a wired network, a local area network (LAN), a wireless network, a wireless LAN (WLAN) network, a radio network, a cellular network, a WiFi network, an IR network, a Bluetooth (BT) network, and the like.

[00036] In some demonstrative aspects, device 102 may be configured to perform and/or to execute one or more operations, modules, processes, procedures and/or the like, e.g., as described herein.

[00037] In some demonstrative aspects, device 102 may include a compiler 160, which may be configured to generate a target code 115, for example, based on a source code 112, e.g., as described below.

[00038] In some demonstrative aspects, compiler 160 may be configured to translate the source code 112 into the target code 115, e.g., as described below.

[00039] In some demonstrative aspects, compiler 160 may include, or may be implemented as, software, a software module, an application, a program, a subroutine,

instructions, an instruction set, computing code, words, values, symbols, and/or the like.

[00040] In some demonstrative aspects, the source code 112 may include computer code written in a source language.

[00041] In some demonstrative aspects, the source language may include a programing language. For example, the source language may include a high-level programming language, for example, such as, C language, C++ language, and/or the like.

[00042] In some demonstrative aspects, the target code 115 may include computer code written in a target language.

[00043] In some demonstrative aspects, the target language may include a low-level language, for example, such as, assembly language, object code, machine code, or the like.

[00044] In some demonstrative aspects, the target code 115 may include one or more object files, e.g., which may create and/or form an executable program.

[00045] In some demonstrative aspects, the executable program may be configured to be executed on a target computer. For example, the target computer may include a specific computer hardware, a specific machine, and/or a specific operating system.

[00046] In some demonstrative aspects, the executable program may be configured to be executed on a processor 180, e.g., as described below.

[00047] In some demonstrative aspects, processor 180 may include a vector processor 180, e.g., as described below. In other aspects, processor 180 may include any other type of processor.

[00048] Some demonstrative aspects are described herein with respect to a compiler, e.g., compiler 160, configured to compile source code 112 into target code 115 configured to be executed by a vector processor 180, e.g., as described below. In other aspects, a compiler, e.g., compiler 160, configured to compile source code 112 into target code 115 configured to be executed by any other type of processor 180.

[00049] In some demonstrative aspects, processor 180 may be implemented as part of device 102.

[00050] In other aspects, processor 180 may be implemented as part of any other

device, e.g., separate from device 102.

[00051] In some demonstrative aspects, vector processor 180 (also referred to as an "array processor") may include a processor, which may be configured to process an entire vector in one instruction, e.g., as described below.

[00052] In other aspects, the executable program may be configured to be executed on any other additional or alternative type of processor.

[00053] In some demonstrative aspects, the vector processor 180 may be designed to support high-performance image and/or vector processing. For example, the vector processor 180 may be configured to processes 1/2/3/4D arrays of fixed point data and/or floating point arrays, e.g., very quickly and/or efficiently.

[00054] In some demonstrative aspects, the vector processor 180 may be configured to process arbitrary data, e.g., structures with pointers to structures. For example, the vector processor 180 may include a scalar processor to compute the non-vector data, for example, assuming the non-vector data is minimal.

[00055] In some demonstrative aspects, compiler 160 may be implemented as a local application to be executed by device 102. For example, memory unit 194 and/or storage unit 195 may store instructions resulting in compiler 160, and/or processor 191 may be configured to execute the instructions resulting in compiler 160 and/or to perform one or more calculations and/or processes of compiler 160, e.g., as described below.

[00056] In other aspects, compiler 160 may include a remote application to be executed by any suitable computing system, e.g., a server 170.

[00057] In some demonstrative aspects, server 170 may include at least a remote server, a web-based server, a cloud server, and/or any other server.

[00058] In some demonstrative aspects, the server 170 may include a suitable memory and/or storage unit 174 having stored thereon instructions resulting in compiler 160, and a suitable processor 171 to execute the instructions, e.g., as descried below.

[00059] In some demonstrative aspects, compiler 160 may include a combination of a remote application and a local application.

[00060] In one example, compiler 160 may be downloaded and/or received by the user of device 102 from another computing system, e.g., server 170, such that compiler

160 may be executed locally by users of device 102. For example, the instructions may be received and stored, e.g., temporarily, in a memory or any suitable short-term memory or buffer of device 102, e.g., prior to being executed by processor 191 of device 102.

[00061] In another example, compiler 160 may include a client-module to be executed locally by device 102, and a server module to be executed by server 170. For example, the client-module may include and/or may be implemented as a local application, a web application, a web site, a web client, e.g., a Hypertext Markup Language (HTML) web application, or the like.

[00062] For example, one or more first operations of compiler 160 may be performed locally, for example, by device 102, and/or one or more second operations of compiler 160 may be performed remotely, for example, by server 170.

[00063] In other aspects, compiler 160 may include, or may be implemented by, any other suitable computing arrangement and/or scheme.

[00064] In some demonstrative aspects, system 100 may include an interface 110, e.g., a user interface, to interface between a user of device 102 and one or more elements of system 100, e.g., compiler 160.

[00065] In some demonstrative aspects, interface 110 may be implemented using any suitable hardware components and/or software components, for example, processors, controllers, memory units, storage units, input units, output units, communication units, operating systems, and/or applications.

[00066] In some aspects, interface 110 may be implemented as part of any suitable module, system, device, or component of system 100.

[00067] In other aspects, interface 110 may be implemented as a separate element of system 100.

[00068] In some demonstrative aspects, interface 110 may be implemented as part of device 102. For example, interface 110 may be associated with and/or included as part of device 102.

[00069] In one example, interface 110 may be implemented, for example, as middleware, and/or as part of any suitable application of device 102. For example,

interface 110 may be implemented as part of compiler 160 and/or as part of an OS of device 102.

[00070] In some demonstrative aspects, interface 110 may be implemented as part of server 170. For example, interface 110 may be associated with and/or included as part of server 170.

[00071] In one example, interface 110 may include, or may be part of a Web-based application, a web-site, a web-page, a plug-in, an ActiveX control, a rich content component, e.g., a Flash or Shockwave component, or the like.

[00072] In some demonstrative aspects, interface 110 may be associated with and/or may include, for example, a gateway (GW) 113 and/or an Application Programming Interface (API) 114, for example, to communicate information and/or communications between elements of system 100 and/or to one or more other, e.g., internal or external, parties, users, applications and/or systems.

[00073] In some aspects, interface 110 may include any suitable Graphic-User-Interface (GUI) 116 and/or any other suitable interface.

[00074] In some demonstrative aspects, interface 110 may be configured to receive the source code 112, for example, from a user of device 102, e.g., via GUI 116, and/or API 114.

[00075] In some demonstrative aspects, interface 110 may be configured to transfer the source code 112, for example, to compiler 160, for example, to generate the target code 115, e.g., as described below.

[00076] Reference is made to Fig. 2, which schematically illustrates a compiler 200, in accordance with some demonstrative aspects. For example, compiler 160 (Fig. 1) may be implement one or more elements of compiler 200, and/or may perform one or more operations and/or functionalities of compiler 200.

[00077] In some demonstrative aspects, as shown in Fig. 2, compiler 200 may be configured to generate a target code 233, for example, by compiling a source code 212 in a source language.

[00078] In some demonstrative aspects, as shown in Fig. 2, compiler 200 may include a front-end 210 configured to receive and analyze the source code 212 in the

source language.

[00079] In some demonstrative aspects, front-end 210 may be configured to generate an intermediate code 213, for example, based on the source code 212.

[00080] In some demonstrative aspects, intermediate code 213 may include a lower level representation of the source code 212.

[00081] In some demonstrative aspects, front-end 210 may be configured to perform, for example, lexical analysis, syntax analysis, semantic analysis, and/or any other additional or alternative type of analysis, of the source code 212.

[00082] In some demonstrative aspects, front-end 210 may be configured to identify errors and/or problems with an outcome of the analysis of the source code 212. For example, front-end 210 may be configured to generate error information, e.g., including error and/or warning messages, for example, which may identify a location in the source code 212, for example, where an error or a problem is detected.

[00083] In some demonstrative aspects, as shown in Fig. 2, compiler 200 may include a middle-end 220 configured to receive and process the intermediate code 213, and to generate an adjusted, e.g., optimized, intermediate code 223.

[00084] In some demonstrative aspects, middle-end 220 may be configured to perform one or more adjustment, e.g., optimizations, to the intermediate code 213, for example, to generate the adjusted intermediate code 223.

[00085] In some demonstrative aspects, middle-end 220 may be configured to perform the one or more optimizations on the intermediate code 213, for example, independent of a type of the target computer to execute the target code 233.

[00086] In some demonstrative aspects, middle-end 220 may be implemented to support use of the optimized intermediate code 223, for example, for different machine types.

[00087] In some demonstrative aspects, middle-end 220 may be configured to optimize the intermediate representation of the intermediate code 223, for example, to improve performance and/or quality of the produced target code 233.

[00088] In some demonstrative aspects, the one or more optimizations of the intermediate code 213, may include, for example, inline expansion, dead-code

elimination, constant propagation, loop transformation, parallelization, and/or the like.

[00089] In some demonstrative aspects, as shown in Fig. 2, compiler 200 may include a back-end 230 configured to receive and process the adjusted intermediate code 213, and to generate the target code 233 based on the adjusted intermediate code 213.

[00090] In some demonstrative aspects, back-end 230 may be configured to perform one or more operations and/or processes, which may be specific for the target computer to execute the target code 233. For example, back-end 230 may be configured to process the optimized intermediate code 213 by applying to the adjusted intermediate code 213 analysis, transformation, and/or optimization operations, which may be configured, for example, based on the target computer to execute the target code 233.

[00091] In some demonstrative aspects, the one or more analysis, transformation, and/or optimization operations applied to the adjusted intermediate code 213 may include, for example, resource and storage decisions, e.g., register allocation, instruction scheduling, and/or the like.

[00092] In some demonstrative aspects, the target code 233 may include target-dependent assembly code, which may be specific to the target computer and/or a target operating system of the target computer, which is to execute the target code 233.

[00093] In some demonstrative aspects, the target code 233 may include target-dependent assembly code for a processor, e.g., vector processor 180 (Fig. 1).

[00094] In some demonstrative aspects, compiler 200 may include a Vector Micro-Code Processor (VMP) Open Computing Language (OpenCL) compiler, e.g., as described below. In other aspects, compiler 200 may include, or may be implemented as part of, any other type of vector processor compiler.

[00095] In some demonstrative aspects, the VMP OpenCL compiler may include a Low Level Virtual Machine (LLVM) based (LLVM-based) compiler, which may be configured according to an LLVM-based compilation scheme, for example, to lower OpenCL C-code to VMP accelerator assembly code, e.g., suitable for execution by vector processor 180 (Fig. 1).

[00096] In some demonstrative aspects, compiler 200 may include one or more technologies, which may be required to compile code to a format suitable for a VMP

architecture, e.g., in addition to open-sourced LLVM compiler passes.

[00097] In some demonstrative aspects, FE 210 may be configured to parse the OpenCL C-code and to translate it, e.g., through an Abstract Syntax Tree (AST), for example, into an LLVM Intermediate Representation (IR).

[00098] In some demonstrative aspects, compiler 200 may include a dedicated API, for example, to detect a correct pattern for compiler pattern matching, for example, suitable for the VMP. For example, the VMP may be configured as a Complex Instruction Set Computer (CISC) machine implementing a very complex Instruction Set Architecture (ISA), which may be hard to target from standard C code. Accordingly, compiler pattern matching may not be able to easily detect the correct pattern, and for this case the compiler may require a dedicated API.

[00099] In some demonstrative aspects, FE 210 may implement one or more vendor extension built-ins, which may target VMP-specific ISA, for example, in addition to standard OpenCL built-ins, which may be optimized to a VMP machine.

[000100] In some demonstrative aspects, FE 210 may be configured to implement OpenCL structures and/or work item functions.

[000101] In some demonstrative aspects, ME 220 may be configured to process LLVM IR code, which may be general and target-independent, for example, although it may include one or more hooks for specific target architectures.

[000102] In some demonstrative aspects, ME 220 may perform one or more custom passes, for example, to support the VMP architecture, e.g., as described below.

[000103] In some demonstrative aspects, ME 220 may be configured to perform one or more operations of a Control Flow Graph (CFG) Linearization analysis, e.g., as described below.

[000104] In some demonstrative aspects, the CFG Linearization analysis may be configured to linearize the code, for example, by converting if-statements to select patterns, for example, in case VMP vector code does not support standard control flow.

[000105] In one example, ME 220 may receive a given code, e.g., as follows:

```
If (x > 0) {
        A = A + 5;
```

```
} else    {
            B = B * 2;
}
```

According to this example, ME 220 may be configured to apply the CFG Linearization analysis to the given code, e.g., as follows:

```
tmpA = A + 5;
tmpB = B * 2;
mask = x > 0;
A = Select mask, tmpA, A
B = Select not mask, tmpB, B
```

Example (1)

[000106] In some demonstrative aspects, ME 220 may be configured to perform one or more operations of an auto-vectorization analysis, e.g., as described below.

[000107] In some demonstrative aspects, the auto-vectorization analysis may be configured to vectorize, e.g., auto-vectorize, a given code, e.g., to utilize vector capabilities of the VMP.

[000108] In some demonstrative aspects, ME 220 may be configured to perform the auto-vectorization analysis, for example, to vectorize code in a scalar form. For example, some or all operations of the auto-vectorization analysis may not be performed, for example, in case the code is already provided in a vectorized form.

[000109] In some demonstrative aspects, for example, in some use cases and/or scenarios, a compiler may not always be able to auto-vectorize a code, for example, due to data dependencies between loop iterations.

[000110] In one example, ME 220 may receive a given code, e.g., as follows:

```
char* a,b,c;
for (int i=0; i < 2048; i++) {
    a[i]=b[i]+c[i];
```

```
}
```

According to this example, ME 220 may be configured to perform the CFG auto-vectorization analysis by applying a first conversion, e.g., as follows:

```
char* a,b,c;
for (int i=0; i < 2048; i+=32) {
    a[i.i+31]=b[i…i+31]+c[i…i+31];
}
```

Example (2a)

For example, ME 220 may be configured to perform the CFG auto-vectorization analysis by applying a second conversion, for example, following the first conversion, e.g., as follows:

```
char32* a,b,c;
for (int i=0; i < 64; i++) {
    a[i]=b[i]+c[i];
}
```

Example (2b)

[000111] In some demonstrative aspects, ME 220 may be configured to perform one or more operations of a Scratch Pad Memory Loop Access Analysis (SPMLAA), e.g., as described below.

[000112] In some demonstrative aspects, the SPMLAA may define Processing Blocks (PB), e.g., that should be outlined and compiled for VMP later.

[000113] In some demonstrative aspects, the processing blocks may include accelerated loops, which may be executed by the vector unit of the VMP.

[000114] In some demonstrative aspects, a PB, e.g., each PB, may include memory references. For example, some or all memory accesses may refer to local memory banks.

[000115] In some demonstrative aspects, the VMP may enable access to memory banks through AGUs, e.g., AGUs 320 as described below with reference to Fig. 3, and Scatter Gather units (SG).

[000116] In some demonstrative aspects, the AGUs may be pre-configured, e.g., before loop execution. For example, a loop trip count may be calculated, e.g., ahead of running a processing block.

[000117] In some demonstrative aspects, image references, e.g., some or all image references, may be created at this stage, and may be followed by calculation of strides and offsets, e.g., per dimension for each reference.

[000118] In some demonstrative aspects, ME 220 may be configured to perform one or more operations of an AGU planner analysis, e.g., as described below.

[000119] In some demonstrative aspects, the AGU Planner analysis may include iterator assignment, which may cover image references, e.g., all image references, from the entire Processing Block.

[000120] In some demonstrative aspects, an iterator may cover a single reference or a group of references.

[000121] In some demonstrative aspects, one or more memory references may be coalesced and/or reuse a same access through shuffle instructions, and/or saving values read from previous iterations.

[000122] In some demonstrative aspects, other memory references, e.g., which have no linear access pattern, may be handled using a Scatter-Gather (SG) unit, which may have a performance penalty, e.g., as it may require maintaining indices and/or masks.

[000123] In some demonstrative aspects, a plan may be configured as an arrangement of iterators in a processing block. For example, a processing block may have multiple plans, e.g., theoretically.

[000124] In some demonstrative aspects, the AGU Planner analysis may be configured to build all possible plans for all PBs, and to select a combination, e.g., a best combination, e.g., from all valid combinations.

[000125] In some demonstrative aspects, a total number of iterators in a valid combination may be limited, e.g., not to exceed a number of available AGUs on a VMP.

[000126] In some demonstrative aspects, one or more parameters, e.g., including stride, width and/or base, may be defined for an iterator, e.g., for each iterator for example, as part of the AGU Planner analysis. For example, min-max ranges for the iterators may be defined in a dimension, e.g., in each dimension, for example, as part of the AGU Planner analysis.

[000127] In some demonstrative aspects, the AGU Planner analysis may be configured to track and evaluate a memory reference, e.g., each memory reference, to an image, e.g., to understand its access pattern.

[000128] In one example, according to Examples 2a/2b, the image 'a' which is the base address, may be accessed with steps of 32 bytes for 64 iterations.

[000129] In some demonstrative aspects, the LLVM may include a scalar evaluation analysis (SCEV), which may compute an access pattern, e.g., to understand every image reference.

[000130] In some demonstrative aspects, ME 220 may utilize masking capabilities of the AGUs, for example, to avoid maintaining an induction variable, which may have a performance penalty.

[000131] In some demonstrative aspects, ME 220 may be configured to perform one or more operations of a rewrite analysis, e.g., as described below.

[000132] In some demonstrative aspects, the rewrite analysis may be configured to transform the code of a processing block, for example, while setting iterators and/or modifying memory access instructions.

[000133] In some demonstrative aspects, setting of the iterators, e.g., of all iterators, may be implemented in IR in target-specific intrinsics. For example, the setting of the iterators may reside in a pre-header of an outermost loop.

[000134] In some demonstrative aspects, the rewrite analysis may include loop-perfectization analysis, e.g., as described below.

[000135] In some demonstrative aspects, the code may be compiled with a target that substantially all calculations should be executed inside the innermost loop.

[000136] For example, the loop-perfectization analysis may hoist instructions, e.g., to move into a loop an operation performed after a last iteration of the loop.

[000137] For example, the loop-perfectization analysis may sink instructions, e.g., to move into a loop an operation performed before a first iteration of the loop.

[000138] For example, the loop-perfectization analysis may hoist instructions and/or sink instructions, for example, such that substantially all instructions are moved from outer loops to the innermost loops.

[000139] For example, the loop-perfectization analysis may be configured to provide a technical solution to support VMP iterators, e.g., to work on perfectly nested loops only.

[000140] For example, the loop-perfectization analysis may result in a situation where there are no instructions between the “for” statements that compose the loop, e.g., to support VMP iterators, which cannot emulate such cases.

[000141] In some demonstrative aspects, the loop-perfectization analysis may be configured to collapse a nested loop into a single collapsed loop.

[000142] In one example, ME 220 may receive a given code, e.g., as follows:

```
for (int i = 0; i < N; i++) {
    int sum = 0;
    for (int j = 0; j < M; j++)
    {
        sum += a[j + stride * i];
    }
        res[i] = sum;
}
```

According to this example, ME 220 may be configured to perform the loop-perfectization analysis to collapse the nested loop in the code to a single collapsed loop, e.g., as follows:

```
for (int k = 0; k < N * M; k++) {
    sum = (k % M == 0 ? 0 : sum);
    sum += a[k % M + stride * ( k / M )];
```

```
        res[k/M] = sum;
    }
```

Example (3)

[000143] In some demonstrative aspects, ME 220 may be configured to perform one or more operations of a Vector Loop Outlining analysis, e.g., as described below.

[000144] In some demonstrative aspects, the Vector Loop Outlining analysis may be configured to divide a code between a scalar subsystem and a vector subsystem, e.g., vector processing block 310 (Fig. 3) and scalar processor 330 (Fig. 3) as described below with reference to Fig. 3.

[000145] In some demonstrative aspects, the VMP accelerator may include the scalar and/or vector subsystems, e.g., as described below. For example, each of the subsystems may have different compute units/processors. Accordingly, a scalar code may be compiled on a scalar compiler, e.g., an SSC compiler, and/or an accelerated vector code may run on the VMP vector processor.

[000146] In some demonstrative aspects, the Vector Loop Outlining analysis may be configured to create a separate function for a loop body of the accelerated vector code. For example, these functions may be marked for the VMP and/or may continue to the VMP backend, for example, while the rest of the code may be compiled by the SSC compiler.

[000147] In some demonstrative aspects, one or more parts of a vector loop, e.g., configuration of the vector unit and/or initialization of vector registers, may be performed by a scalar unit. However, these parts may be performed in a later stage, for example, by performing backpatching into the scalar code, e.g., as the scalar code may still be in LLVM IR before processing by the SSC compiler.

[000148] In some demonstrative aspects, BE 230 may be configured to translate the LLVM IR into machine instructions. For example, the BE 230 may not be target agnostic and may be familiar with target-specific architecture and optimizations, e.g., compared to ME 220, which may be agnostic to a target-specific architecture.

[000149] In some demonstrative aspects, BE 230 may be configured to perform one or more analyses, which may be specific to a target machine, e.g., a VMP machine, to which the code is lowered, e.g., although BE 230 may use common LLVM.

[000150] In some demonstrative aspects, BE 230 may be configured to perform one or more operations of an instruction lowering analysis, e.g., as described below.

[000151] In some demonstrative aspects, the instruction lowering analysis may be configured to translate LLVM IR into target-specific instructions Machine IR (MIR), for example, by translating the LLVM IR into a Directed Acyclic Graph (DAG).

[000152] In some demonstrative aspects, the DAG may go through a legalization process of instructions, for example, based on the data types and/or VMP instructions, which may be supported by a VMP HW.

[000153] In some demonstrative aspects, the instruction lowering analysis may be configured to perform a process of pattern-matching, e.g., after the legalization process of instructions, for example, to lower a node, e.g., each node, in the DAG, for example, into a VMP-specific machine instruction.

[000154] In some demonstrative aspects, the instruction lowering analysis may be configured to generate the MIR, for example, after the process of pattern-matching.

[000155] In some demonstrative aspects, the instruction lowering analysis may be configured to lower the instruction according to machine Application Binary Interface (ABI) and/or calling conventions.

[000156] In some demonstrative aspects, BE 230 may be configured to perform one or more operations of a unit balancing analysis, e.g., as described below.

[000157] In some demonstrative aspects, the unit balancing analysis may be configured to balance instructions between VMP compute units, e.g., data processing units 316 (Fig. 3) as described below with reference to Fig. 3.

[000158] In some demonstrative aspects, the unit balancing analysis may be familiar with some or all available arithmetic transformations, and/or may perform transformations according to an optimal algorithm.

[000159] In some demonstrative aspects, BE 230 may be configured to perform one or more operations of a modulo scheduler (pipeliner) analysis, e.g., as described below.

[000160] In some demonstrative aspects, the pipeliner may be configured to schedule the instructions according to one or more constraints, e.g., data dependency, resource bottlenecks and/or any other constrains, for example, using Swing Modulo Scheduling (SMS) heuristics and/or any other additional and/or alternative heuristic.

[000161] In some demonstrative aspects, the pipeliner may be configured to schedule a set, e.g., an Initiation Interval (II), of Very Long Instruction Word (VLIW) instructions that the program will iterate on, e.g., during a steady state.

[000162] In some demonstrative aspects, a performance metric, which may be based on a number of cycles a typical loop may execute, may be measured, e.g., as follows:

*(Size of Input data in bytes) * II / (Bytes consumed/produced every iteration)*

[000163] In some demonstrative aspects, the pipeliner may try to minimize the II, e.g., as much as possible, for example, to improve performance.

[000164] In some demonstrative aspects, the pipeliner may be configured to calculate a minimum II, and to schedule accordingly. For example, if the pipeliner fails the scheduling, the pipeliner may try to increase the II and retry scheduling, e.g., until a predefined II threshold is violated.

[000165] In some demonstrative aspects, BE 230 may be configured to perform one or more operations of a register allocation analysis, e.g., as described below.

[000166] In some demonstrative aspects, the register allocation analysis may be configured to attempt to assign a register in an efficient, e.g., optimal, way.

[000167] In some demonstrative aspects, the register allocation analysis may assign values to bypass vector registers, general purpose vector registers, and/or scalar registers.

[000168] In some demonstrative aspects, the values may include private variables, constants, and/or values that are rotated across iterations.

[000169] In some demonstrative aspects, the register allocation analysis may implement an optimal heuristic that suites one or more VMP register file (regfile) constraints. For example, in some use cases, the register allocation analysis may not use a standard LLVM register allocation.

[000170] In some demonstrative aspects, in some cases, the register allocation analysis may fail, which may mean that the loop cannot be compiled. Accordingly, the register allocation analysis may implement a retry mechanism, which may go back to the modulo scheduler and may attempt to reschedule the loop, e.g., with an increased initiation interval. For example, increasing the initiation interval may reduce register pressure, and/or may support compilation of the vector loop, e.g., in many cases.

[000171] In some demonstrative aspects, BE 230 may be configured to perform one or more operations of an SSC configuration analysis, e.g., as described below.

[000172] In some demonstrative aspects, the SSC configuration analysis may be configured to set a configuration to execute the kernel, e.g., the AGU configuration.

[000173] In some demonstrative aspects, the SSC configuration analysis may be performed at a late stage, for example, due to configurations calculated after legalization, the register allocation analysis, and/or the modulo scheduling analysis.

[000174] In some demonstrative aspects, the SSC configuration analysis may include a Zero Overhead Loop (ZOL) mechanism in the vector loop. For example, the ZOL mechanism may configure a loop trip count based on an access pattern of the memory references in the loop, for example, to avoid running instructions that check the loop exit condition every iteration.

[000175] In some demonstrative aspects, a VMP Compilation Flow may include one or more, e.g., a few, steps, which may be invoked during the compilation flow in a test library (testlib), e.g., a wrapper script for compilation, execution, and/or program testing. For example, these steps may be performed outside of the LLVM Compiler.

[000176] In some demonstrative aspects, a PCB Hardware Description Language (PHDL) simulator may be implemented to perform one or more roles of an assembler, encoder, and/or linker.

[000177] In some demonstrative aspects, compiler 200 may be configured to provide a technical solution to support robustness, which may enable compilation of a vast selection of loops, with HW limitations. For example, compiler 200 may be configured to support a technical solution, which may not create verification errors.

[000178] In some demonstrative aspects, compiler 200 may be configured to provide a technical solution to support programmability, which may provide a user an ability to express code in multiple ways, which may compile correctly to the VMP architecture.

[000179] In some demonstrative aspects, compiler 200 may be configured to provide a technical solution to support an improved user-experience, which may allow the user capability to debug and/or profile code. For example, the improved user-experience may provide informative error messages, report tools, and/or a profiler.

[000180] In some demonstrative aspects, compiler 200 may be configured to provide a technical solution to support improved performance, for example, to optimize a VMP assembly code and/or iterator accesses, which may lead to a faster execution. For example, improved performance may be achieved through high utilization of the compute units and usage of its complex CISC.

[000181] Reference is made to Fig. 3, which schematically illustrates a vector processor 300, in accordance with some demonstrative aspects. For example, vector processor 180 (Fig. 1) may be implement one or more elements of vector processor 300, and/or may perform one or more operations and/or functionalities of vector processor 300.

[000182] In some demonstrative aspects, vector processor 300 may include a Vector Microcode Processor (VMP).

[000183] In some demonstrative aspects, vector processor 300 may include a Wide Vector machine, for example, supporting Very Long Instruction Word (VLIW) architectures, and/or Single Instruction/Multiple Data (SIMD) architectures.

[000184] In some demonstrative aspects, vector processor 300 may be configured to provide a technical solution to support high performance for short integral types, which may be common, for example, in computer-vision and/or deep-learning algorithms.

[000185] In other aspects, vector processor 300 may include any other type of vector processor, and/or may be configured to support any other additional or alternative functionalities.

[000186] In some demonstrative aspects, as shown in Fig. 3, vector processor 300 may include a vector processing block (vector processor) 310, a scalar processor 330, and a Direct Memory Access (DMA) 340, e.g., as described below.

[000187] In some demonstrative aspects, as shown in Fig. 3, vector processing block 310 may be configured to process, e.g., efficiently process, image data and/or vector data. For example, the vector processing block 310 may be configured to use vector computation units, for example, to speed up computations.

[000188] In some demonstrative aspects, scalar processor 330 may be configured to perform scalar computations. For example, the scalar processor 330 may be used as a "glue logic" for programs including vector computations. For example, some, e.g., even most, of the computation of the programs may be performed by the vector processing block 310. However, several tasks, for example, some essential tasks, e.g., scalar computations, may be performed by the scalar processor 330.

[000189] In some demonstrative aspects, the DMA 340 may be configured to interface with one or more memory elements in a chip including vector processor 300.

[000190] In some demonstrative aspects, the DMA 340 may be configured to read inputs from a main memory, and/or write outputs to the main memory.

[000191] In some demonstrative aspects, the scalar processor 330 and the vector processing block 310 may use respective local memories to process data.

[000192] In some demonstrative aspects, as shown in Fig. 3, vector processor 300 may include a fetcher and decoder 350, which may be configured to control the scalar processor 330 and/or the vector processing block 310.

[000193] In some demonstrative aspects, operations of the scalar processor 330 and/or the vector processing block 310 may be triggered by instructions stored in a program memory 352.

[000194] In some demonstrative aspects, the DMA 340 may be configured to transfer data, for example, in parallel with the execution of the program instructions in memory 352.

[000195] In some demonstrative aspects, DMA 340 may be controlled by software, e.g., via configuration registers, for example, rather than instructions, and, accordingly, may be considered as a second "thread" of execution in vector processor 300.

[000196] In some demonstrative aspects, the scalar processor 330, the vector processing block 310, and/or the DMA 340 may include one or more data processing units, for example, a set of data processing units, e.g., as described below.

[000197] In some demonstrative aspects, the data processing units may include hardware configured to preform computations, e.g., an Arithmetic Logic Unit (ALU).

[000198] In one example, a data processing unit may be configured to add numbers, and/or to store the numbers in a memory.

[000199] In some demonstrative aspects, the data processing units may be controlled by commands, e.g., encoded in the program memory 352 and/or in configuration registers. For example, the configuration registers may be memory mapped, and may be written by the memory store commands of the scalar processor 330.

[000200] In some demonstrative aspects, the scalar processor 330, the vector processing block 310, and/or the DMA 340 may include a state configuration including a set of registers and memories, e.g., as described below.

[000201] In some demonstrative aspects, as shown in Fig. 3, vector processor block 310 may include a set of vector memories 312, which may be configured, for example, to store data to be processed by vector processor block 310.

[000202] In some demonstrative aspects, as shown in Fig. 3, vector processor block 310 may include a set of vector registers 314, which may be configured, for example, to be used in data processing by vector processor block 310.

[000203] In some demonstrative aspects, the scalar processor 330, the vector processing block 310, and/or the DMA 340 may be associated with a set of memory maps.

[000204] In some demonstrative aspects, a memory map may include a set of addresses accessible by a data processing unit, which may load and/or store data from/to registers and memories.

[000205] In some demonstrative aspects, as shown in Fig. 3, the vector processing block 310 may include a plurality of Address Generation Units (AGUs) 320, which may include addresses accessible to them, e.g., in one or more of memories 312.

[000206] In some demonstrative aspects, as shown in Fig. 3, vector processor block 310 may include a plurality of data processing units 316, e.g., as described below.

[000207] In some demonstrative aspects, data processing units 316 may be configured to process commands, e.g., including several numbers at a time. In one example, a command may include 8 numbers. In another example, a command may include 4 numbers, 16 numbers, or any other count of numbers.

[000208] In some demonstrative aspects, two or more data processing units 316 may be used simultaneously. In one example, data processing units 316 may process and execute a plurality of different command, e.g., 3 different commands, for example, including 8 numbers, at a throughout of a single cycle.

[000209] In some demonstrative aspects, data processing units 316 may be asymmetrical. For example, first and second data processing units 316 may support different commands. For example, addition may be performed by a first data processing unit 316, and/or multiplication may be performed by a second data processing unit 316. For example, both operations may be performed by one or more additional other data processing units 316.

[000210] In some demonstrative aspects, data processing units 316 may be configured to support arithmetic operations for many combinations of input & output data types.

[000211] In some demonstrative aspects, data processing units 316 may be configured to support one or more operations, which may be less common. For example, processing units 316 may support operations working with a Look Up Table (LUT) of vector processor 300, and/or any other operations.

[000212] In some demonstrative aspects, data processing units 316 may be configured to support efficient computation of non-linear functions, histograms, and/or random data access, e.g., which may be useful to implement algorithms like image scaling, Hough transforms, and/or any other algorithms.

[000213] In some demonstrative aspects, vector memories 312 may include, for example, memory banks having a size of 16K or any other size, which may be accessed at a same cycle.

[000214] In one example, a maximal memory access size may be 64 bits. According to this example, a peak throughput may be 256 bits, e.g., 64x4 = 256. For example, high

memory bandwidth may be implemented to utilize computation capabilities of the data processing units 316.

[000215] In one example, two data processing units 316 may support 16 8-bit multiply & accumulate operations (MACs) per cycle. According to this example, the two data processing units 316 may not be useful, for example, in case the input numbers are not fetched at this speed, and/or there isn't exactly 256 bits of input, e.g., *16x8x2 = 256*.

[000216] In some demonstrative aspects, AGUs 320 may be configured to perform memory access operations, e.g., loading and storing data from/to vector memories 314.

[000217] In some demonstrative aspects, AGUs 320 may be configured to compute addresses of input and output data items, for example, to handle I/O to utilize the data processing units 316, e.g., in case sheer bandwidth is not enough.

[000218] In some demonstrative aspects, AGUs 320 may be configured to compute the addresses of the input and/or output data items, for example, based on configuration registers written by the scalar processor 330, for example, before a block of vector commands, e.g., a loop, is entered.

[000219] For example, AGUs 320 may be configured to write an image base pointer, a width, a height and/or a stride to the configuration registers, for example, in order to iterate over an image.

[000220] In some demonstrative aspects, AGUs 320 may be configured to handle addressing, e.g., all addressing, for example, to provide a technical solution in which data processing units 316 may not have the burden of incrementing pointers or counters in a loop, and/or the burden to check for end-of-row conditions, e.g., to zero a counter in the loop.

[000221] In some demonstrative aspects, as shown in Fig. 3, AGUs 320 may include 4 AGUs, and, accordingly, four memories 312 may be accessed at a same cycle. In other aspects, any other count of AGUs 32 may be implemented.

[000222] In some demonstrative aspects, AGUs 320 may not be "tied" to memory banks 312. For example, an AGU 320, e.g., each AGU 320, may access a memory bank 312, e.g., every memory bank 312, for example, as long as two or more AGUs 320 do not try to access the same memory bank 312 at the same cycle.

[000223] In some demonstrative aspects, vector registers 314 may be configured to support communication between the data processing units 316 and AGUs 320.

[000224] In one example, a total number of vector registers 314 may be 28, which may be divided into several subsets, e.g., based on their function. For example, a first subset of vector registers 314 may be used for inputs/outputs, e.g., of all data processing units 316 and/or AGUs 320; and/or a second subset of vector registers 314 may not be used for outputs of some operations, e.g., most operations, and may be used for one or more other operations, e.g., to store loop-invariant inputs.

[000225] In some demonstrative aspects, a data processing unit 316, e.g., each data processing unit 316, may have one or more registers to host an output of a last executed operation, e.g., which may be fed as inputs to other data processing units 316. For example, these registers may "bypass" the vector registers 314, and may work faster than writing these outputs to first set of vector registers 314.

[000226] In some demonstrative aspects, fetcher and decoder 350 may be configured to support low-overhead vector loops, e.g., very low overhead vector loops (also referred to as “zero-overhead vector loops”), for example, where there may be no need to check a termination (exit) condition of a vector loop during an execution of the vector loop.

[000227] For example, a termination (exit) condition may be signaled by an AGU 320, for example, when the AGU 320 finishes iterating over a configured memory region.

[000228] For example, fetcher and decoder 350 may quit the loop, for example, when the AGU 320 signals the termination condition.

[000229] For example, the scalar processor 330 may be utilized to configure the loop parameters, e.g., first & last instructions and/or the exit condition.

[000230] In one example, vector loops may be utilized, for example, together with high memory bandwidth and/or cheap addressing, for example, to solve a control and data flow problem, for example, to provide a technical solution to allow the data processing units 316 to process data, e.g., without substantially additional overhead.

[000231] In some demonstrative aspects, scalar processor 330 may be configured to provide one or more functionalities, which may be complementary to those of the vector

processing block 310. For example, a large portion, e.g., most, of the work in a vector program may be performed by the data processing units 316. For example, the scalar processor 330 may be utilized, for example, for "gluing" together the various blocks of vector code of the vector program.

[000232] In some demonstrative aspects, scalar processor 330 may be implemented separately from vector processing block 310. In other aspects, scalar processor 330 may be configured to share one or more components and/or functionalities with vector processing block 310.

[000233] In some demonstrative aspects, scalar processor 330 may be configured to perform operations, which may not be suitable for execution on vector processing block 310.

[000234] For example, scalar processor 330 may be utilized to execute 32 bit C programs. For example, scalar processor 330 may be configured to support 1, 2, and/or 4 byte data types of C code, and/or some or all arithmetic operators of C code.

[000235] For example, scalar processor 330 may be configured to provide a technical solution to perform operations that cannot be executed on vector processing block 310, for example, without using a full-blown CPU.

[000236] In some demonstrative aspects, scalar processor 330 may include a scalar data memory 332, e.g., having a size of 16K or any other size, which may be configured to store data, e.g., variables used by the scalar parts of a program.

[000237] For example, scalar processor 330 may store local and/or global variables declared by portable C code, which may be allocated to scalar data memory by a compiler, e.g., compiler 200 (Fig. 2).

[000238] In some demonstrative aspects, as shown in Fig. 3, scalar processor 330 may include, or may be associated with, a set of vector registers 334, which may be used in data processing performed by the scalar processor 330.

[000239] In some demonstrative aspects, scalar processor 330 may be associated with a scalar memory map, which may support scalar processor 330 in accessing substantially all states of vector processor 300. For example, the scalar processor 330 may configure the vector units and/or the DMA channels via the scalar memory map.

[000240] In some demonstrative aspects, scalar processor 330 may not be allowed to access one or more block control registers, which may be used by external processors to run and debug vector programs.

[000241] In some demonstrative aspects, DMA 340 may be configured to communicate with one or more other components of a chip implementing the vector processor 300, for example, via main memory. For example, DMA 340 may be configured to transfer blocks of data, e.g., large, contiguous, blocks of data, for example, to support the scalar processor 330 and/or the vector processing block, which may manipulate data stored in the local memories. For example, a vector program may be able to read data from the main chip memory using DMA 340.

[000242] In some demonstrative aspects, DMA 340 may be configured to communicate with other elements of the chip, for example, via a plurality of DMA channels, e.g., 8 DMA channels or any other count of DMA channels. For example, a DMA channel, e.g., each DMA channel, may be capable of transferring a rectangular patch from the local memories to the main chip memory, or vice versa. In other aspects, the DMA channel may transfer any other type of data block between the local memories and the main chip memory.

[000243] In some demonstrative aspects, a rectangular patch may be defined by a base pointer, a width, a height, and astride.

[000244] For example, at peak throughput, 8 bytes per cycle may be transferred, however, there may be overheads for each patch and/or for each row in a patch.

[000245] In some demonstrative aspects, DMA 340 may be configured to transfer data, for example, in parallel with computations, e.g., via the plurality of DMA channels, for example, as long as executed commands do not access a local memory involved in the transfer.

[000246] In one example, as all channels may access the same memory bus, using several channels to implement a transfer may not save I/O cycles, e.g., compared to the case when a single channel is used. However, the plurality of DMA channels may be utilized to schedule several transfers and execute them in parallel with computations. This may be advantageous, for example, compared to a single channel, which may not allow scheduling a second transfer before completion of the first transfer.

[000247] In some demonstrative aspects, DMA 340 may be associated with a memory map, which may support the DMA channels in accessing vector memories and/or the scalar data. For example, access to the vector memories may be performed in parallel with computations. For example, access to the scalar data may usually not be allowed in parallel, e.g., as the scalar processor 330 may be involved in almost any sensible program, and may likely access it's local variables while the transfer is performed, which may lead to a memory contention with the active DMA channel.

[000248] In some demonstrative aspects, DMA 340 may be configured to provide a technical solution to support parallelization of I/O and computations. For example, a program performing computations may not have to wait for I/O, for example, in case these computations may run fast by vector processing block 310.

[000249] In some demonstrative aspects, an external processor, e.g., a CPU, may be configured to initiate execution of a program on vector processor 300. For example, vector processor 300 may remain idle, e.g., as long as program execution is not initiated.

[000250] In some demonstrative aspects, the external processor may be configured to debug the program, e.g., execute a single step at a time, halt when the program reaches breakpoints, and/or inspect contents of registers and memories storing the program variables.

[000251] In some demonstrative aspects, an external memory map may be implemented to support the external processor in controlling the vector processor 300 and/or debugging the program, for example, by writing to control registers of the vector processor 300.

[000252] In some demonstrative aspects, the external memory map may be implemented by a superset of the scalar memory map. For example, this implementation may make all registers and memories defined by the architecture of the vector processor 300 accessible to a debugger back-end running on the external processor.

[000253] In some demonstrative aspects, the vector processor 300 may raise an interrupt signal, for example, when the vector processor 300 terminates a program.

[000254] In some demonstrative aspects, the interrupt signal may be used, for example to implement a driver to maintain a queue of programs scheduled for execution

by the vector processor 300, and/or to launch a new program, e.g., by the external processor, for example, upon the completion of a previously executed program.

[000255] Referring back to Fig. 1, in some demonstrative aspects, compiler 160 may be configured to generate the target code 115 configured, for example, to utilize registers of a processor, for example, a vector processor, e.g., vector processor 180, according to a register allocation scheme, e.g., as described below.

[000256] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to utilize a reduced number of allocated registers for executing a program by a processor, for example, a vector processor, e.g., as described below.

[000257] In one example, the register allocation scheme may be configured to provide a technical solution to utilize a reduced number of allocated registers, which may be allocated from the plurality of vector registers 314 (Fig. 3), for execution of a program by vector processor 300 (Fig. 3), e.g., as described below.

[000258] In some demonstrative aspects, a compiler, e.g., compiler 160, may be configured to generate target code, e.g., the target code 115, which may be configured to utilize registers of a vector processor, e.g., vector processor 189, according to a register allocation scheme, e.g., as described below.

[000259] In other aspects, a compiler, e.g., compiler 160, may be configured to generate target code, e.g., the target code 115, which may be configured to utilize registers of any other suitable type of processor, e.g., any other suitable type of processor, according to the register allocation scheme, e.g., as described below.

[000260] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution improve, e.g., to optimize, an allocation of registers to execute the executable program, e.g., as described below.

[000261] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to support an improved allocation, for example, an efficient allocation, e.g., an optimized allocation, of registers to execute the executable program, e.g., as described below.

[000262] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to utilize a reduced number, for example, an

optimized number, e.g., a minimal number, of allocated registers to execute the executable program, e.g., as described below.

[000263] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to improve performance of the executable program, for example, by reducing the number of allocated registers to execute the executable program, e.g., as described below.

[000264] In some demonstrative aspects, there may be a need to provide a technical solution to efficiently allocate vector registers of a vector processor for execution of a program, for example, in order to reduce a number of the allocated vector registers, e.g., as described below.

[000265] For example, a number of physical registers implemented by a chip including a processor, e.g., a vector processor or any other processor, may be limited, for example, according to a design and/or layout of the chip. Accordingly, a number of vector registers implemented by the vector processor may be limited by the number of physical registers on the chip implementing the vector processor.

[000266] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to reduce the number of the allocated registers, for example, for CPUs, e.g., vector processors, having limited storage capabilities, e.g., a limited register pool, and/or for processors, e.g., vector processors, which have limited support of, or do not support, memory spill/fill operations, e.g., to store live values.

[000267] For example, processors having no fill/spill capabilities and/or limited storage capabilities may be forced to use compute resources instead.

[000268] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to reduce the number of the allocated registers, for example, to avoid, or even eliminate, the use of these extra compute resources.

[000269] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to reduce a register pressure for execution of a program by a processor, for example, a vector processor, or any other processor.

[000270] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to reduce the register pressure, for example, by reducing the number of allocated registers for execution of the program, e.g., as described below.

[000271] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to support efficient execution of programs, e.g., complex programs, which may be sensitive to register pressure. For example, for some programs, a register pressure issue may be a bottleneck, and, accordingly, the register pressure may affect performance.

[000272] In some demonstrative aspects, the register allocation scheme may be configured to provide a technical solution to reduce the number of allocated registers for execution of a program, for example, while providing a suitable allocation of registers, e.g., vector registers, for execution of the program, e.g., as described below.

[000273] In some demonstrative aspects, complier 160 may be configured to process a given instruction scheduling, e.g., based on the source code 112, and to generate target code 115, which may be configured to utilize a reduced number of allocated registers, for example, for a successful register allocation, e.g., as described below.

[000274] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115 configured to utilize data processing units, e.g., ALUs, of a processor, e.g., a vector processor, for storage of variables of the executable program, e.g., live values, e.g., as described below.

[000275] In some demonstrative aspects, compiler 160 may be configured generate the target code 115 configured to utilize the data processing units of the vector processor for storage of one or more variables of the executable program, e.g., live values, for example, instead of storing these variables in one or more registers, e.g., as described below.

[000276] In one example, compiler 160 may be configured to generate the target code 115 configured to utilize one or more of the data processing units 316 (Fig. 3) for storage, e.g., temporal storage, of one or more variables of the executable program, e.g., live values, for example, instead of storing one or more of these variables in one or more vector registers 314 (Fig. 3), e.g., as described below.

[000277] In some demonstrative aspects, compiler 160 may be configured generate the target code 115 configured to exploit an internal state of ALUs, for example, to store the live values in the ALUs, for example, instead of storing these live values in physical registers, e.g., as described below.

[000278] In some demonstrative aspects, compiler 160 may be configured generate the target code 115 configured to exploit a latency of performing instructions by the ALUs, for example, for storing the live values in the ALUs, e.g., as described below.

[000279] In some demonstrative aspects, compiler 160 may be configured generate the target code 115 configured to store live values of variables in the ALUs, for example, by live interval splitting, e.g., as described below.

[000280] In some demonstrative aspects, the live interval splitting may include splitting a live range (interval) of a live value of a variable, for example, into a first live interval, e.g., where the live value is stored by an ALU, and a second live interval, e.g., where the live value is stored in a physical vector register, e.g., as described below.

[000281] In some demonstrative aspects, a live range (interval) of a live value of a variable may be split more than once, e.g., to provide more than two live intervals. For example, a count of the live intervals may be increased, for example, to provide more cycles during which registers may be available, e.g., as described below.

[000282] In some demonstrative aspects, a live range of a variable may include a range of cycles of the executable program, for example, between a first cycle, e.g., which includes a first use and/or a production of the variable, and a second cycle, e.g., which includes a second use, e.g., subsequent to the first use, of the variable, e.g., as described below.

[000283] In some demonstrative aspects, compiler 160 may be configured to identify one or more cycles ("unused variable cycles") in the live range of a variable, during which the variable is live and not being used, e.g., as described below.

[000284] In some demonstrative aspects, compiler 160 may be configured to allocate in the target code 115 one or more no operation (no-op) instructions to be applied to the variable, for example, to store the live value of the variable by an ALU, e.g., as described below.

[000285] In some demonstrative aspects, compiler 160 may be configured to allocate in the target code one or more no-op instructions to be executed by an ALU, for example, such that the live value of the variable may be stored by the ALU executing the one or more no-op instructions, e.g., as described below.

[000286] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115 including the one or more no-op instructions, which may be configured to exploit a latency of the ALU for executing instructions, for example, such that the live value of the variable may be temporarily stored by the ALU, e.g., as described below.

[000287] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115 including the one or more no-op instructions, which may be configured to provide a technical solution to temporarily store the live value of the variable by the ALU executing the one or more no-op instructions, for example, instead of storing the live value of the variable in a vector register, e.g., as described below.

[000288] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115 including the one or more no-op instructions to be applied to the variable, for example, even without substantially affecting throughput of the executable program, e.g., as described below.

[000289] In some demonstrative aspects, a no-op instruction may include an instruction, which may be configured to cause the ALU to perform a sequence of operations, which may be executed by the ALU, and may result in maintaining the live value of the variable during one or more cycles, e.g., as described below.

[000290] In some demonstrative aspects, the no-op instruction may include an instruction, which may be configured to cause the ALU to perform a sequence of operations, for example, including a load operation to load the live value of the variable from a vector register, an operation to be applied to the live value of the variable, e.g., without changing the live value of the variable, and a store operation to store the live value of the variable back in the same register, e.g., as described below.

[000291] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115 including one or more no-op instructions, which, when executed by the ALU, may cause the ALU to execute the sequence of operations of the no-op

instructions within a number of cycles (latency cycles), which may be based, for example, on a latency of the ALU for performing instructions, e.g., as described below.

[000292] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115 including one or more no-op instructions, which, when executed by the ALU, may result in storage of the live value of the variable by the ALU for the duration of the latency cycles.

[000293] In one example, the one or more no-op instructions may include an addition-with-zero instruction.

[000294] In another example, the one or more no-op instructions may include a shift-by-zero instruction, e.g., a left-shift-by-zero instruction, or a right-shift-by-zero instruction.

[000295] In another example, the one or more no-op instructions may include a multiplication-by-one instruction.

[000296] In another example, the one or more no-op instructions may include any other additional and/or alternative instruction, which may be configured to, when executed by an ALU, cause the ALU to store a live value of a variable by the ALU, e.g., for the duration of one or more latency cycles.

[000297] In some demonstrative aspects, compiler 160 may be configured to compile a source code 112 into a target code 115, which may be configured for execution by a target processor 180, e.g., as described below.

[000298] In some demonstrative aspects, compiler 160 may be configured to compile the source code 112 including OpenCL code, e.g., as described below. In other aspects, compiler 160 may be configured to compile any other type of source code 112.

[000299] In some demonstrative aspects, compiler 160 may be configured to compile the source code 112 into the target code 115, for example, according to an LLVM-based compilation scheme. In other aspects, any other additional or alternative compilation scheme may be utilized.

[000300] In some demonstrative aspects, compiler 160 may be configured to compile the source code 112 into the target code 115, which may be configured, for example, for execution by a VLIW SIMD target processor.

[000301] In some demonstrative aspects, compiler 160 may be configured to compile the source code 112 into the target code 115, which may be configured, for example, for execution by a target vector processor.

[000302] In other aspects, the compiler 160 may be configured to compile the source code 112 into the target code 115, which may be configured for execution by any other additional or alternative type of processor.

[000303] In some demonstrative aspects, target code 115 may be configured for execution by the target processor 180 in a plurality of execution cycles, for example, including a first execution cycle, a second execution cycle after the first execution cycle, and a third execution cycle after the second execution cycle, e.g., as described below.

[000304] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115 including, for example, one or more no operation (no-op) instructions, which may be configured to maintain a first variable live, for example, such that a value, e.g., a live value, of the first variable is to be available in a register of the target processor, for example, at the first execution cycle and at the third execution cycle, e.g., as described below.

[000305] In some demonstrative aspects, compiler 160 may configure the target code 115 to include a first instruction to be applied, for example, to the value, e.g., the live value, of the first variable in the register at the first execution cycle, e.g., as described below.

[000306] In some demonstrative aspects, compiler 160 may configure the target code 115 to include a second instruction to be applied, for example, to a value of a second variable in the register at the second execution cycle, e.g., as described below.

[000307] In some demonstrative aspects, compiler 160 may configure the target code 115 to include a third instruction to be applied, for example, to the value, e.g., the live value, of the first variable in the register at the third execution cycle, e.g., as described below.

[000308] In some demonstrative aspects, compiler 160 may be configured to output the target code 115, for example, in a form executable by the target processor 180.

[000309] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115 configured, for example, for execution by a Very Long Instruction Word (VLIW) Single Instruction/Multiple Data (SIMD) target processor, e.g., processor 180.

[000310] In other aspects, compiler 160 may be configured to generate the target code 115 configured, for example, for execution by any other suitable type of processor.

[000311] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115, for example, based on the source code 112 including Open Computing Language (OpenCL) code.

[000312] In other aspects, compiler 160 may be configured to generate the target code 115, for example, based on the source code 112 including any other suitable type of code.

[000313] In some demonstrative aspects, compiler 160 may be configured to compile the source code 112 into the target code 115, for example, according to a Low Level Virtual Machine (LLVM) based (LLVM-based) compilation scheme.

[000314] In other aspects, compiler 160 may be configured to compile the source code 112 into the target code 115 according to any other suitable compilation scheme.

[000315] In some demonstrative aspects, compiler 160 may be configured to configure the one or more no-op instructions, for example, to cause the target processor 180 to apply to the value, e.g., the live value, of the first variable one or more operations, which may be configured, for example, to maintain the value, e.g., the live value, of the first variable unchanged between the first execution cycle and the third execution cycle, e.g., as described below.

[000316] In some demonstrative aspects, compiler 160 may be configured to configure the one or more no-op instructions, for example, to cause a data processing unit of the target processor 180, e.g., a data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), to temporarily maintain the value, e.g., the live value, of the first variable internally at the data processing unit, for example, between the first execution cycle and the third execution cycle, e.g., as described below.

[000317] In some demonstrative aspects, compiler 160 may be configured to configure the one or more no-op instructions to include, for example, an addition with zero instruction, e.g., as describe below.

[000318] In some demonstrative aspects, compiler 160 may be configured to configure the one or more no-op instructions to include, for example, a shift-by-zero instruction, e.g., as described below.

[000319] In some demonstrative aspects, compiler 160 may be configured to configure the one or more no-op instructions to include, for example, a multiplication-by-one instruction, e.g., as described below.

[000320] In other aspects, compiler 160 may be configured to configure the one or more no-op instructions to include any other suitable additional or alternative type of no-op instruction.

[000321] In some demonstrative aspects, compiler 160 may be configured to configure the one or more no-op instructions, for example, to cause a data processing unit of the target processor 180, e.g., a data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), to initiate loading of the value, e.g., the live value, of the first variable from the register at the first execution cycle, e.g., as described below.

[000322] In some demonstrative aspects, compiler 160 may be configured to configure the one or more no-op instructions, for example, to cause the data processing unit of the target processor 180, e.g., a data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), to initiate storing of the value, e.g., the live value, of the first variable in the register at a later execution cycle, for example, before the third execution cycle, e.g., as described below.

[000323] In some demonstrative aspects, compiler 160 may be configured to generate the one or more no-op instructions to include a plurality of no-op instructions, e.g., as described below.

[000324] In some demonstrative aspects, compiler 160 may be configured to generate a first-in-order no-op instruction of the plurality of no-op instructions, which may be configured, for example, to cause a data processing unit of the target processor 180, e.g., a data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), to load the value, e.g., the live value, of the first variable from the register, e.g., at the first

execution cycle, and to apply to the value, e.g., the live value, of the first variable a first-in-order operation, for example, configured to maintain the value, e.g., the live value, of the first variable unchanged, e.g., as described below.

[000325] In some demonstrative aspects, compiler 160 may be configured to generate a last-in-order no-op instruction of the plurality of no-op instructions, which may be configured, for example, to cause the data processing unit of the target processor 180, e.g., a data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), to apply to an output of a previous no-op instruction a last-in-order operation, which may be configured, for example, to maintain the value, e.g., the live value, of the first variable unchanged, and to store the value, e.g., the live value, of the first variable in the register, e.g., as described below.

[000326] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115, which may be configured, for example, such that execution of a last-in-order no-op instruction of the one or more no-op instructions is to be initiated by a data processing unit of the target processor 180, e.g., a data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), for example, at a last-in-order no-op execution cycle, e.g., prior to the third execution cycle, e.g., as described below.

[000327] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115, which may be configured, for example, such that a distance between the last-in-order no-op execution cycle and the third execution cycle may be based, for example, on a latency of the data processing unit of the target processor 180, e.g., a data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), e.g., as described below.

[000328] In some demonstrative aspects, compiler 160 may be configured to configure the one or more no-op instructions, for example, based on a latency of a data processing unit of the target processor 180, e.g., a data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), e.g., as described below.

[000329] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115, which may be configured, for example, such that the first instruction is to be executed, for example, by a first data processing unit of the target

processor 180, e.g., a first data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), e.g., as described below.

[000330] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115, which may be configured, for example, such that the one or more no-op instructions are to be executed, for example, by a second data processing unit of the target processor 180, e.g., a first data processing unit 316 (Fig. 3) of vector processor 300 (Fig. 3), e.g., as described below.

[000331] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115, which may be configured, for example, such that execution of a first-in-order no-op instruction of the one or more no-op instructions is to be initiated by the second data processing unit, for example, at the first execution cycle, e.g., as described below.

[000332] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115, which may be configured, for example, such that the second instruction is to be executed, for example, by the second data processing unit, e.g., as described below.

[000333] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115, which may be configured, for example, such that the third instruction is to be executed, for example, by the first data processing unit, e.g., as described below.

[000334] In some demonstrative aspects, the first data processing unit of the target processor 180 may include a first ALU, and/or the second data processing unit of the target processor 180 may include a second ALU, e.g., as described below. In other aspects, any other additional or alternative type of data processing unit may be implemented.

[000335] In some demonstrative aspects, compiler 160 may be configured to identify a plurality of live ranges corresponding to a respective plurality of variables based on the source code 112, e.g., as described below.

[000336] In some demonstrative aspects, compiler 160 may be configured to identify as the first variable an identified variable, for example, having a live range including

one or more unused execution cycles in which the identified variable is not used, e.g., as described below.

[000337] In some demonstrative aspects, compiler 160 may be configured to identify the identified variable as the first variable, for example, based on a determination that a count of consecutive unused execution cycles in the live range of the identified variable is greater than a predefined threshold, e.g., as described below.

[000338] In some demonstrative aspects, compiler 160 may be configured to configure the one or more no-op instructions, for example, to maintain the first variable live during the one or more unused execution cycles, e.g., as described below.

[000339] In some demonstrative aspects, compiler 160 may be configured to allocate the register to store the value of the second variable, for example, at an unused execution cycle of the one or more unused execution cycles, e.g., as described below.

[000340] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115 based on VLIW instructions, which may be utilized to call a plurality of different instructions during a same cycle. For example, the plurality of the instructions may include a no-op instruction, which may be executed in parallel with other instructions, for example, such that performance may not be affected by the addition of the no-op instruction.

[000341] In some demonstrative aspects, compiler 160 may be configured to receive the source code 112 including an instruction scheduling for an executed program, e.g., as described below.

[000342] In some demonstrative aspects, compiler 160 may be configured to identify a live range of one or more variables in the executed program, e.g., as described below.

[000343] In some demonstrative aspects, compiler 160 may be configured to identify one or more cycles (“unused cycles”) in a live range of a variable.

[000344] For example, the unused cycles may include cycles during which the variable is live but not in use, e.g., as described below.

[000345] In some demonstrative aspects, compiler 160 may be configured to determine an identified data processing unit, e.g., an ALU, which has a latency when

performing no-op instructions, and which is available during the one or more unused cycles, e.g., as described below.

[000346] In some demonstrative aspects, compiler 160 may be configured to insert one or more no-op instructions to be applied to the variable by the identified data processing unit, for example, during the one or more unused cycles, e.g., as described below.

[000347] In some demonstrative aspects, causing the identified data processing unit to perform the no-op operation on the live variable may result in the identified data processing unit temporarily storing the variable during the one or more unused cycles, for example, according to the latency of the identified data processing unit, e.g., as described below.

[000348] In some demonstrative aspects, using the identified data processing unit to store the variable during the one or more unused cycles may provide a technical solution to free a register, e.g., a vector register, for example, during the one or more unused cycles, e.g., as described below.

[000349] In some demonstrative aspects, the free register, e.g., the free vector register, may be utilized, for example, to store one or more other live values of other variables, for example, during the unused cycles. As a result, a required number of registers, e.g., vector registers, for execution of the program may be reduced.

[000350] In some demonstrative aspects, compiler 160 may be configured to generate no-op instructions, which may be configured to utilize instructions with different latency, for example, to provide a technical solution to free one or more register cycles, e.g., as may be needed.

[000351] In some demonstrative aspects, compiler 160 may be configured to generate the target code 115, for example, including one or more no-op instructions, for example, to allocate register vectors, for example, in an efficient manner, e.g., as described below.

[000352] In some demonstrative aspects, compiler 160 may receive a source code 112 of an executed program.

[000353] In some demonstrative aspects, source code 112 may include an instruction scheduling, which may be configured to calculate an expression based on a first variable, denoted *a*, and a second variable, denoted *b*, e.g., as follows:

$$(a \cdot 2 + b \cdot 3) \cdot a \quad (1)$$

[000354] In some demonstrative aspects, an expression result of the Expression 1 may be stored in a result memory address, denoted "*res_add*".

[000355] In some demonstrative aspects, the variable *a* may be stored in a memory address, denoted *a_add*, and the variable *b* may be stored in a memory address, denoted *b_add.*

[000356] In some demonstrative aspects, compiler 160 may be configured to compile the source code 112 for a processor, for example, a vector processor, including a first ALU and a second ALU, which may be configured to perform add and multiply instructions, for example, with a latency of 2 cycles. For example, the processor, e.g., the vector processor, may have a memory unit with a latency of 1 cycle for a memory access operation, e.g., a store operation to store a value to the memory unit, or a load operation to load a value from the memory unit.

[000357] In some demonstrative aspects, execution of the Expression 1 may be implemented according to the following instruction scheduling:

Cycle	**ALU1**	**ALU2**	**Load/Store**
0			a = load(a_add)
1	a_mul_2 = mul(a,2)		b = load(b_add)
2		b_mul_3 = mul(b,3)	
3			
4	sum = add(a_mul_2, b_mul_3)		
5			
6	res = mul(sum, a)		
7			

8			store(res_add,res)

TABLE (1)

[000358] As shown in Table 1, one or more first operations may be performed by the first ALU, and one or more second operation may be performed by the second ALU.

[000359] As shown in Table 1, the second ALU may not perform any operations during the cycle 1 and the cycle 3.

[000360] As shown in Table 1, the execution of the Expression 1 may be performed during 9 cycles.

[000361] In one example, three registers, denoted *R0, R1,* and *R2,* may be utilized to execute the instructions of Table 1, e.g., as follows:

Cycle	**R0**	**R1**	**R2**
0	Don't Care	Don't Care	Don't Care
1	a	Don't Care	Don't Care
2	a	Don't Care	b
3	a	a_mul_2	Don't Care
4	a	a_mul_2	b_mul_3
5	a	Don't Care	Don't Care
6	a	Sum	Don't Care
7	Don't Care	Don't Care	Don't Care
8	res	Don't Care	Don't Care

TABLE (2)

[000362] As shown in Table 2, the register *R0* may be required to store the variable *a* during cycles 1-6, and to store the expression result during the cycle 8.

[000363] As shown in Table 2, the register *R2* may be required to store the multiplication result, denoted *b_mul_3*, of the product of variable *b* and 3, during the cycle 4.

[000364] As shown in Table 2, the register *R2* may be required during the cycles 2 and 4, while the register R1 may be required to store the value *a_mul_2* during cycles 3,4, and to store the value sum in cycle 6.

[000365] In some demonstrative aspects, compiler 160 may be configured to identify live ranges of variables in the instruction scheduling according to Table 1, e.g., as follows:

Value	**Live Range in Cycles [Start, End] (Including) and Assigned Register**
a	[1, 6]: R0
b	[2,2]: R2
a_mul_2	[3, 4]: R1
b_mul_3	[4, 4]: R2
sum	[6, 6]: R1
res	[8, 8]: R0

TABLE (3)

[000366] As shown in Table 3, the variable *a* may have a live range between the cycle 1 and the cycle 6, while the variable *a* may not be used during some of these cycles, e.g., the cycles 2-5, as shown in Table 1.

[000367] In some demonstrative aspects, compiler 160 may identify that the variable *a* is unused during one or more unused cycles, e.g., cycles 2-5, within the live range between cycle 1 and cycle 6.

[000368] In some demonstrative aspects, compiler 160 may identify that the second ALU (ALU2) is free during the cycle 1 and the cycle 3.

[000369] In some demonstrative aspects, compiler 160 may insert a first no-op instruction, denoted *a_1*, having a latency of 2 cycles, to be performed by the second ALU (ALU2) at the cycle 1, for example, in order to free the register *R0* during the cycle 2, e.g., as described below.

[000370] In some demonstrative aspects, compiler 160 may insert a second no-op instruction, denoted *a_2*, having a latency of 2 cycles, to be performed by the second ALU (ALU2) at the cycle 3, for example, in order to free the register *R0* during the cycle 4, e.g., as described below.

[000371] In some demonstrative aspects, compiler 160 may assign the register *R0* to store the multiplication result *b_mul_3*, for example, during the cycle 4. For example, the register *R0*, which may be free at cycle 4, may be used to store the multiplication result *b_mul_3*, e.g., instead of the register *R2*. Accordingly, the register *R2* may become redundant, e.g., as no other operation requires the use of register R2 according to Table 2.

[000372] In some demonstrative aspects, compiler 160 may determine target code 115 based on an updated instruction scheduling including the first and second no-op instructions, e.g., as follows:

Cycle	**ALU1**	**ALU2**	**Load/Store**
0			a = load(a_add)
1	a_mul_2 = mul(a,2)	a_1 = add(a, 0)	b = load(b_add)
2		b_mul_3 = mul(b,3)	
3		a_2 = add(a_1, 0)	
4	sum = add(a_mul_2, b_mul_3)		

5			
6	res = mul(sum, a_2)		
7			
8		store(res_add, res)	

TABLE (4)

[000373] In some demonstrative aspects, the variables of the updated instruction scheduling may have updated live ranges, which may be different from the live ranges of Table 3, e.g., as follows.

Value	**Live Range in Cycles [Start, End] (Including) and Assigned Register**	**Notes**
a	[1], [3], [5, 6]: R0	R0 is available for use in cycle 2, 4
b	[2,2]: R0	
a_mul_2	[3, 4]: R1	
b_mul_3	[4, 4]: R0	
sum	[6, 6]: R1	
res	[8, 8]: R0	

TABLE (5)

[000374] As shown in Table 5, the variable *a* may be assigned to the register *R0* at the cycle 1, the cycle 3, the cycle 5, and the cycle 6.

[000375] As shown in Table 5, the variable *a* may not be assigned to the register *R0* at the cycle 2 and the cycle 4, e.g., as the variable *a* may be temporarily stored by the second ALU, e.g., when executing the no-op instructions.

[000376] As shown in Table 5, the multiplication result *b_mul_3* may be assigned to the register *R0* at cycle 4. As a result, the register R2 may become redundant.

[000377] In some demonstrative aspects, the updated instruction scheduling may be configured to allocate two vector registers, e.g., the registers *R0* and *R1*, while the third register *R2* may not be required, e.g., as follows.

Cycle	**R0**	**R1**	**R2**
0	Don't Care	Don't Care	
1	a	Don't Care	
2	b	Don't Care	
3	a	a_mul_2	
4	b_mul_3	a_mul_2	
5	a	Don't Care	
6	a	sum	
7	Don't Care	Don't Care	
8	res	Don't Care	

TABLE (6)

[000378] As shown in Table 6, compiler 160 may assign the register *R0* to store the variable *a* at the cycle 1, the cycle 3, the cycle 5 and the cycle 6; to store the multiplication result *b_mul_3* at the cycle 4, and to store the result at cycle 8.

[000379] As shown in Table 6, the register *R2* may become redundant.

[000380] In some demonstrative aspects, a number of cycles of the executed program according to the instruction set of Table 1 may be equal to a number of cycles of the executed program according to the instruction set of Table 4, e.g., 9 cycles. However, the number of allocated registers may be reduced by the updated instruction scheduling, e.g., from three registers to two registers. Accordingly, the instruction set of Table 4 may be implemented to provide a technical solution with improved, e.g., optimized, performance, e.g., as follows:

Metric	**Example Without Optimization**	**Example With Optimization**
Total Run Time (Cycles)	9	9
Number of Used Registers	3	2

TABLE (7)

[000381] In some demonstrative aspects, as shown in Table 7, the register allocation scheme may be implemented to provide a technical solution to save a vector register for usage, for example, without affecting performance.

[000382] For example, in some use cases and/or scenarios, a scheduling for a program may result in an unsuccessful register allocation, e.g., due to a limited number of registers.

[000383] In one example, an attempt to schedule execution of the Expression 1 according to the instruction scheduling of Table 1 may result in an unsuccessful register allocation, e.g., if only two registers are available. One option to address this situation may be to relax the instruction scheduling, e.g., in attempt to reduce the number of live variables sharing the same execution cycles. However, this option may result in degraded performance.

[000384] In some demonstrative aspects, execution of the Expression 1 according to the register allocation scheme described above, e.g., using the instruction scheduling of Table 4, may provide a technical solution to support successful register allocation, e.g., even if only two registers are available, for example, while avoiding the performance degradation resulting from the instruction scheduling of Table 1.

[000385] Reference is made to Fig. 4, which schematically illustrates a method of compiling code for a processor. For example, one or more operations of the method of Fig. 4 may be performed by a system, e.g., system 100 (Fig. 1); a device, e.g., device 102 (Fig. 1); a server, e.g., server 170 (Fig. 1); and/or a compiler, e.g., compiler 160 (Fig. 1), and/or compiler 200 (Fig. 2).

[000386] In some demonstrative aspects, as indicated at block 402, the method may include observing live ranges, for example, live rages above a predefined threshold, e.g., relatively large live ranges, where a variable "x" may not be used for a period of time, e.g., a relatively long time. For example, compiler 160 (Fig. 1) may identify one or more unused cycles in the live range, e.g., as described above.

[000387] In some demonstrative aspects, as indicated at block 404, the method may include checking if there is a free ALU in the unused cycles that can execute a no-op operation. For example. compiler 160 (Fig. 1) may check to identify a data processing unit of the processor, which is available during the one or more unused cycles, e.g., as described above.

[000388] In some demonstrative aspects, as indicated at block 406, the method may include inserting one or more no-op instructions operating on the variable "x", which may return the variable "x" to a same register after one or more latency cycles during the unused cycles. For example. compiler 160 (Fig. 1) may generate the target code 115 based on the one or more no-op instructions to be executed during the live range of the variable, e.g., as described above.

[000389] In some demonstrative aspects, one or more operations of the method of Fig. 4 may be repeated, e.g., for each variable in a source code, for example, to generate target code 115 according to a register allocation scheme, which may be configured to reduce a number of required registers for execution of the source code. For example, reducing the number of required registers may provide a technical solution to support efficient use of CPUs with a limited number of registers and/or no support of memory spill/fill.

[000390] Reference is made to Fig. 5, which schematically illustrates a method of compiling code for a processor. For example, one or more operations of the method of Fig. 5 may be performed by a system, e.g., system 100 (Fig. 1); a device, e.g., device

102 (Fig. 1); a server, e.g., server 170 (Fig. 1); and/or a compiler, e.g., compiler 160 (Fig. 1), and/or compiler 200 (Fig. 2).

[000391] In some demonstrative aspects, as indicated at block 502, the method may include compiling a source code into a target code. For example, the target code may be configured for execution by a target processor in a plurality of execution cycles, e.g., including a first execution cycle, a second execution cycle after the first execution cycle, and a third execution cycle after the second execution cycle. For example, the target code may include one or more no operation (no-op) instructions, which may be configured, for example, to maintain a first variable live, for example, such that a value, e.g., a live value, of the first variable is to be available in a register of the target processor at the first execution cycle and at the third execution cycle. For example, the target code may include a first instruction to be applied to the value, e.g., the live value, of the first variable in the register at the first execution cycle, a second instruction to be applied to a value of a second variable in the register at the second execution cycle, and/or a third instruction to be applied to the value, e.g., the live value, of the first variable in the register at the third execution cycle. For example, compiler 160 (Fig. 1) may be configured generate the target code 115 including the one or more no no-op instructions, e.g., as descried above.

[000392] In some demonstrative aspects, as indicated at block 504, the method may include outputting the target code. For example, compiler 160 (Fig. 1) may be configured to output the target code 115 (Fig. 1), e.g., as described above.

[000393] Reference is made to Fig. 6, which schematically illustrates a product of manufacture 600, in accordance with some demonstrative aspects. Product 600 may include one or more tangible computer-readable ("machine-readable") non-transitory storage media 602, which may include computer-executable instructions, e.g., implemented by logic 604, operable to, when executed by at least one computer processor, enable the at least one computer processor to implement one or more operations at device 102 (Fig. 1), server 170 (Fig. 1), and/or compiler 160 (Fig. 1), to cause device 102 (Fig. 1), server 170 (Fig. 1), and/or compiler 160 (Fig. 1) to perform, trigger and/or implement one or more operations and/or functionalities, and/or to perform, trigger and/or implement one or more operations and/or functionalities described with reference to the Figs. 1-5, and/or one or more operations described

herein. The phrases "non-transitory machine-readable medium" and "computer-readable non-transitory storage media" may be directed to include all computer-readable media, with the sole exception being a transitory propagating signal.

[000394] In some demonstrative aspects, product 600 and/or machine-readable storage media 602 may include one or more types of computer-readable storage media capable of storing data, including volatile memory, non-volatile memory, removable or non-removable memory, erasable or non-erasable memory, writeable or re-writeable memory, and the like. For example, machine-readable storage media 602 may include, RAM, DRAM, Double-Data-Rate DRAM (DDR-DRAM), SDRAM, static RAM (SRAM), ROM, programmable ROM (PROM), erasable programmable ROM (EPROM), electrically erasable programmable ROM (EEPROM), flash memory (e.g., NOR or NAND flash memory), content addressable memory (CAM), polymer memory, phase-change memory, ferroelectric memory, silicon-oxide-nitride-oxide-silicon (SONOS) memory, a disk, a hard drive, and the like. The computer-readable storage media may include any suitable media involved with downloading or transferring a computer program from a remote computer to a requesting computer carried by data signals embodied in a carrier wave or other propagation medium through a communication link, e.g., a modem, radio or network connection.

[000395] In some demonstrative aspects, logic 604 may include instructions, data, and/or code, which, if executed by a machine, may cause the machine to perform a method, process and/or operations as described herein. The machine may include, for example, any suitable processing platform, computing platform, computing device, processing device, computing system, processing system, computer, processor, or the like, and may be implemented using any suitable combination of hardware, software, firmware, and the like.

[000396] In some demonstrative aspects, logic 604 may include, or may be implemented as, software, a software module, an application, a program, a subroutine, instructions, an instruction set, computing code, words, values, symbols, and the like. The instructions may include any suitable type of code, such as source code, compiled code, interpreted code, executable code, static code, dynamic code, and the like. The instructions may be implemented according to a predefined computer language, manner or syntax, for instructing a processor to perform a certain function. The instructions

may be implemented using any suitable high-level, low-level, object-oriented, visual, compiled and/or interpreted programming language, machine code, and the like.

EXAMPLES

[000397] The following examples pertain to further aspects.

[000398] Example 1 includes a product comprising one or more tangible computer-readable non-transitory storage media comprising computer-executable instructions operable to, when executed by at least one computer processor, enable the at least one computer processor to cause a compiler to compile a source code into a target code, the target code configured for execution by a target processor in a plurality of execution cycles comprising a first execution cycle, a second execution cycle after the first execution cycle, and a third execution cycle after the second execution cycle, wherein the target code comprises one or more no operation (no-op) instructions configured to maintain a first variable live such that a value of the first variable is to be available in a register of the target processor at the first execution cycle and at the third execution cycle, wherein the target code comprises a first instruction to be applied to the value of the first variable in the register at the first execution cycle, a second instruction to be applied to a value of a second variable in the register at the second execution cycle, and a third instruction to be applied to the value of the first variable in the register at the third execution cycle; and output the target code.

[000399] Example 2 includes the subject matter of Example 1, and optionally, wherein the one or more no-op instructions are configured to cause the target processor to apply to the value of the first variable one or more operations configured to maintain the value of the first variable unchanged between the first execution cycle and the third execution cycle.

[000400] Example 3 includes the subject matter of Example 1 or 2, and optionally, wherein the one or more no-op instructions are configured to cause a data processing unit of the target processor to temporarily maintain the value of the first variable internally at the data processing unit between the first execution cycle and the third execution cycle.

[000401] Example 4 includes the subject matter of any one of Examples 1-3, and optionally, wherein the one or more no-op instructions are configured to cause a data processing unit of the target processor to initiate loading of the value of the first variable from the register at the first execution cycle, and to initiate storing of the value of the first variable in the register at a later execution cycle before the third execution cycle.

[000402] Example 5 includes the subject matter of any one of Examples 1-4, and optionally, wherein the one or more no-op instructions comprises a plurality of no-op instructions comprising a first-in-order no-op instruction and a last-in-order no-op instruction, wherein the first-in-order no-op instruction is configured to cause a data processing unit of the target processor to load the value of the first variable from the register at the first execution cycle and to apply to the value of the first variable a first-in-order operation configured to maintain the value of the first variable unchanged, wherein the last-in-order no-op instruction is configured to cause the data processing unit of the target processor to apply to an output of a previous no-op instruction a last-in-order operation configured to maintain the value of the first variable unchanged, and to store the value of the first variable in the register.

[000403] Example 6 includes the subject matter of any one of Examples 1-5, and optionally, wherein the target code is configured such that execution of a last-in-order no-op instruction of the one or more no-op instructions is to be initiated by a data processing unit of the processor at a last-in-order no-op execution cycle prior to the third execution cycle, wherein a distance between the last-in-order no-op execution cycle and the third execution cycle is based on a latency of the data processing unit.

[000404] Example 7 includes the subject matter of any one of Examples 1-6, and optionally, wherein the instructions, when executed, cause the compiler to configure the one or more no-op instructions based on a latency of a data processing unit of the target processor.

[000405] Example 8 includes the subject matter of any one of Examples 1-7, and optionally, wherein the target code is configured such that the first instruction is to be executed by a first data processing unit of the target processor, and the one or more no-op instructions are to be executed by a second data processing unit of the target processor.

[000406] Example 9 includes the subject matter of Example 8, and optionally, wherein the target code is configured such that execution of a first-in-order no-op instruction of the one or more no-op instructions is to be initiated by the second data processing unit at the first execution cycle.

[000407] Example 10 includes the subject matter of Example 8 or 9, and optionally, wherein the target code is configured such that the second instruction is to be executed by the second data processing unit.

[000408] Example 11 includes the subject matter of any one of Examples 8-10, and optionally, wherein the target code is configured such that the third instruction is to be executed by the first data processing unit.

[000409] Example 12 includes the subject matter of any one of Examples 8-11, and optionally, wherein the first data processing unit comprises a first Arithmetic Logic Unit (ALU), and the second data processing unit comprises a second ALU.

[000410] Example 13 includes the subject matter of any one of Examples 1-12, and optionally, wherein the instructions, when executed, cause the compiler to identify a plurality of live ranges corresponding to a respective plurality of variables based on the source code, and to identify as the first variable an identified variable having a live range including one or more unused execution cycles in which the identified variable is not used.

[000411] Example 14 includes the subject matter of Example 13, and optionally, wherein the instructions, when executed, cause the compiler to identify the identified variable as the first variable based on a determination that a count of consecutive unused execution cycles in the live range of the identified variable is greater than a predefined threshold.

[000412] Example 15 includes the subject matter of Example 13 or 14, and optionally, wherein the instructions, when executed, cause the compiler to configure the one or more no-op instructions to maintain the first variable live during the one or more unused execution cycles.

[000413] Example 16 includes the subject matter of any one of Examples 13-15, and optionally, wherein the instructions, when executed, cause the compiler to allocate the

register to store the value of the second variable at an unused execution cycle of the one or more unused execution cycles.

[000414] Example 17 includes the subject matter of any one of Examples 1-16, and optionally, wherein the one or more no-op instructions comprise at least one of an addition with zero instruction, a shift-by-zero instruction, or a multiplication-by-one instruction.

[000415] Example 18 includes the subject matter of any one of Examples 1-17, and optionally, wherein the source code comprises Open Computing Language (OpenCL) code.

[000416] Example 19 includes the subject matter of any one of Examples 1-18, and optionally, wherein the instructions, when executed, cause the compiler to compile the source code into the target code according to a Low Level Virtual Machine (LLVM) based (LLVM-based) compilation scheme.

[000417] Example 20 includes the subject matter of any one of Examples 1-19, and optionally, wherein the target code is configured for execution by a Very Long Instruction Word (VLIW) Single Instruction/Multiple Data (SIMD) target processor.

[000418] Example 21 includes the subject matter of any one of Examples 1-20, and optionally, wherein the target code is configured for execution by a target vector processor.

[000419] Example 22 includes a compiler configured to perform any of the described operations of any of Examples 1-21.

[000420] Example 23 includes a computing device configured to perform any of the described operations of any of Examples 1-21.

[000421] Example 24 includes a computing system comprising at least one memory to store instructions; and at least one processor to retrieve instructions from the memory and execute the instructions to cause the computing system to perform any of the described operations of any of Examples 1-21.

[000422] Example 25 includes a computing system comprising a compiler to generate target code according to any of the described operations of any of Examples 1-21, and a processor to execute the target code.

[000423] Example 26 comprises an apparatus comprising means for executing any of the described operations of any of Examples 1-21.

[000424] Example 27 comprises an apparatus comprising: a memory interface; and processing circuitry configured to: perform any of the described operations of any of Examples 1-21.

[000425] Example 28 comprises a method comprising any of the described operations of any of Examples 1-21.

[000426] Functions, operations, components and/or features described herein with reference to one or more aspects, may be combined with, or may be utilized in combination with, one or more other functions, operations, components and/or features described herein with reference to one or more other aspects, or vice versa.

[000427] While certain features have been illustrated and described herein, many modifications, substitutions, changes, and equivalents may occur to those skilled in the art. It is, therefore, to be understood that the appended claims are intended to cover all such modifications and changes as fall within the true spirit of the disclosure.

CLAIMS

What is claimed is:

1. A product comprising one or more tangible computer-readable non-transitory storage media comprising computer-executable instructions operable to, when executed by at least one processor, enable the at least one processor to cause a compiler to:

compile a source code into a target code, the target code configured for execution by a target processor in a plurality of execution cycles comprising a first execution cycle, a second execution cycle after the first execution cycle, and a third execution cycle after the second execution cycle, wherein the target code comprises one or more no operation (no-op) instructions configured to maintain a first variable live such that a value of the first variable is to be available in a register of the target processor at the first execution cycle and at the third execution cycle, wherein the target code comprises a first instruction to be applied to the value of the first variable in the register at the first execution cycle, a second instruction to be applied to a value of a second variable in the register at the second execution cycle, and a third instruction to be applied to the value of the first variable in the register at the third execution cycle; and

output the target code.

2. The product of claim 1, wherein the one or more no-op instructions are configured to cause the target processor to apply to the value of the first variable one or more operations configured to maintain the value of the first variable unchanged between the first execution cycle and the third execution cycle.

3. The product of claim 1, wherein the one or more no-op instructions are configured to cause a data processing unit of the target processor to temporarily maintain the value of the first variable internally at the data processing unit between the first execution cycle and the third execution cycle.

4. The product of claim 1, wherein the one or more no-op instructions are configured to cause a data processing unit of the target processor to initiate loading of the value of the first variable from the register at the first execution cycle, and to initiate storing of the value of the first variable in the register at a later execution cycle before the third execution cycle.

5. The product of claim 1, wherein the one or more no-op instructions comprises a plurality of no-op instructions comprising a first-in-order no-op instruction and a last-in-order no-op instruction, wherein the first-in-order no-op instruction is configured to cause a data processing unit of the target processor to load the value of the first variable from the register at the first execution cycle and to apply to the value of the first variable a first-in-order operation configured to maintain the value of the first variable unchanged, wherein the last-in-order no-op instruction is configured to cause the data processing unit of the target processor to apply to an output of a previous no-op instruction a last-in-order operation configured to maintain the value of the first variable unchanged, and to store the value of the first variable in the register.

6. The product of claim 1, wherein the target code is configured such that execution of a last-in-order no-op instruction of the one or more no-op instructions is to be initiated by a data processing unit of the processor at a last-in-order no-op execution cycle prior to the third execution cycle, wherein a distance between the last-in-order no-op execution cycle and the third execution cycle is based on a latency of the data processing unit.

7. The product of claim 1, wherein the instructions, when executed, cause the compiler to configure the one or more no-op instructions based on a latency of a data processing unit of the target processor.

8. The product of claim 1, wherein the target code is configured such that the first instruction is to be executed by a first data processing unit of the target processor, and the one or more no-op instructions are to be executed by a second data processing unit of the target processor.

9. The product of claim 8, wherein the target code is configured such that execution of a first-in-order no-op instruction of the one or more no-op instructions is to be initiated by the second data processing unit at the first execution cycle.

10. The product of claim 8, wherein the target code is configured such that the second instruction is to be executed by the second data processing unit.

11. The product of claim 8, wherein the target code is configured such that the third instruction is to be executed by the first data processing unit.

12. The product of claim 8, wherein the first data processing unit comprises a first Arithmetic Logic Unit (ALU), and the second data processing unit comprises a second ALU.

13. The product of any one of claims 1-12, wherein the instructions, when executed, cause the compiler to identify a plurality of live ranges corresponding to a respective plurality of variables based on the source code, and to identify as the first variable an identified variable having a live range including one or more unused execution cycles in which the identified variable is not used.

14. The product of claim 13, wherein the instructions, when executed, cause the compiler to identify the identified variable as the first variable based on a determination that a count of consecutive unused execution cycles in the live range of the identified variable is greater than a predefined threshold.

15. The product of claim 13, wherein the instructions, when executed, cause the compiler to configure the one or more no-op instructions to maintain the first variable live during the one or more unused execution cycles.

16. The product of claim 13, wherein the instructions, when executed, cause the compiler to allocate the register to store the value of the second variable at an unused execution cycle of the one or more unused execution cycles.

17. The product of any one of claims 1-12, wherein the one or more no-op instructions comprise at least one of an addition with zero instruction, a shift-by-zero instruction, or a multiplication-by-one instruction.

18. The product of any one of claims 1-12, wherein the source code comprises Open Computing Language (OpenCL) code.

19. The product of any one of claims 1-12, wherein the instructions, when executed, cause the compiler to compile the source code into the target code according to a Low Level Virtual Machine (LLVM) based (LLVM-based) compilation scheme.

20. The product of any one of claims 1-12, wherein the target code is configured for execution by a Very Long Instruction Word (VLIW) Single Instruction/Multiple Data (SIMD) target processor.

21. The product of any one of claims 1-12, wherein the target code is configured for execution by a target vector processor.

22. A computing system comprising:

at least one memory to store instructions; and

at least one processor to retrieve the instructions from the memory and to execute the instructions to cause the computing system to:

compile a source code into a target code, the target code configured for execution by a target processor in a plurality of execution cycles comprising a first execution cycle, a second execution cycle after the first execution cycle, and a third execution cycle after the second execution cycle, wherein the target code comprises one or more no operation (no-op) instructions configured to maintain a first variable live such that a value of the first variable is to be available in a register of the target processor at the first execution cycle and at the third execution cycle, wherein the target code comprises a first instruction to be applied to the value of the first variable in the register at the first execution cycle, a second instruction to be applied to a value of a second variable in the register at the second execution cycle, and a third instruction to be applied to the value of the first variable in the register at the third execution cycle; and

output the target code.

23. The computing system of claim 22, wherein the one or more no-op instructions are configured to cause the target processor to apply to the value of the first variable one or more operations configured to maintain the value of the first variable unchanged between the first execution cycle and the third execution cycle.

24. The computing system of claim 22, wherein the one or more no-op instructions are configured to cause a data processing unit of the target processor to initiate loading of the value of the first variable from the register at the first execution cycle, and to

initiate storing of the value of the first variable in the register at a later execution cycle before the third execution cycle.

25. The computing system of claim 22 comprising the target processor.

26. A method comprising:

compiling a source code into a target code, the target code configured for execution by a target processor in a plurality of execution cycles comprising a first execution cycle, a second execution cycle after the first execution cycle, and a third execution cycle after the second execution cycle, wherein the target code comprises one or more no operation (no-op) instructions configured to maintain a first variable live such that a value of the first variable is to be available in a register of the target processor at the first execution cycle and at the third execution cycle, wherein the target code comprises a first instruction to be applied to the value of the first variable in the register at the first execution cycle, a second instruction to be applied to a value of a second variable in the register at the second execution cycle, and a third instruction to be applied to the value of the first variable in the register at the third execution cycle; and

outputting the target code.

27. The method of claim 26, wherein the one or more no-op instructions comprise at least one of an addition with zero instruction, a shift-by-zero instruction, or a multiplication-by-one instruction.

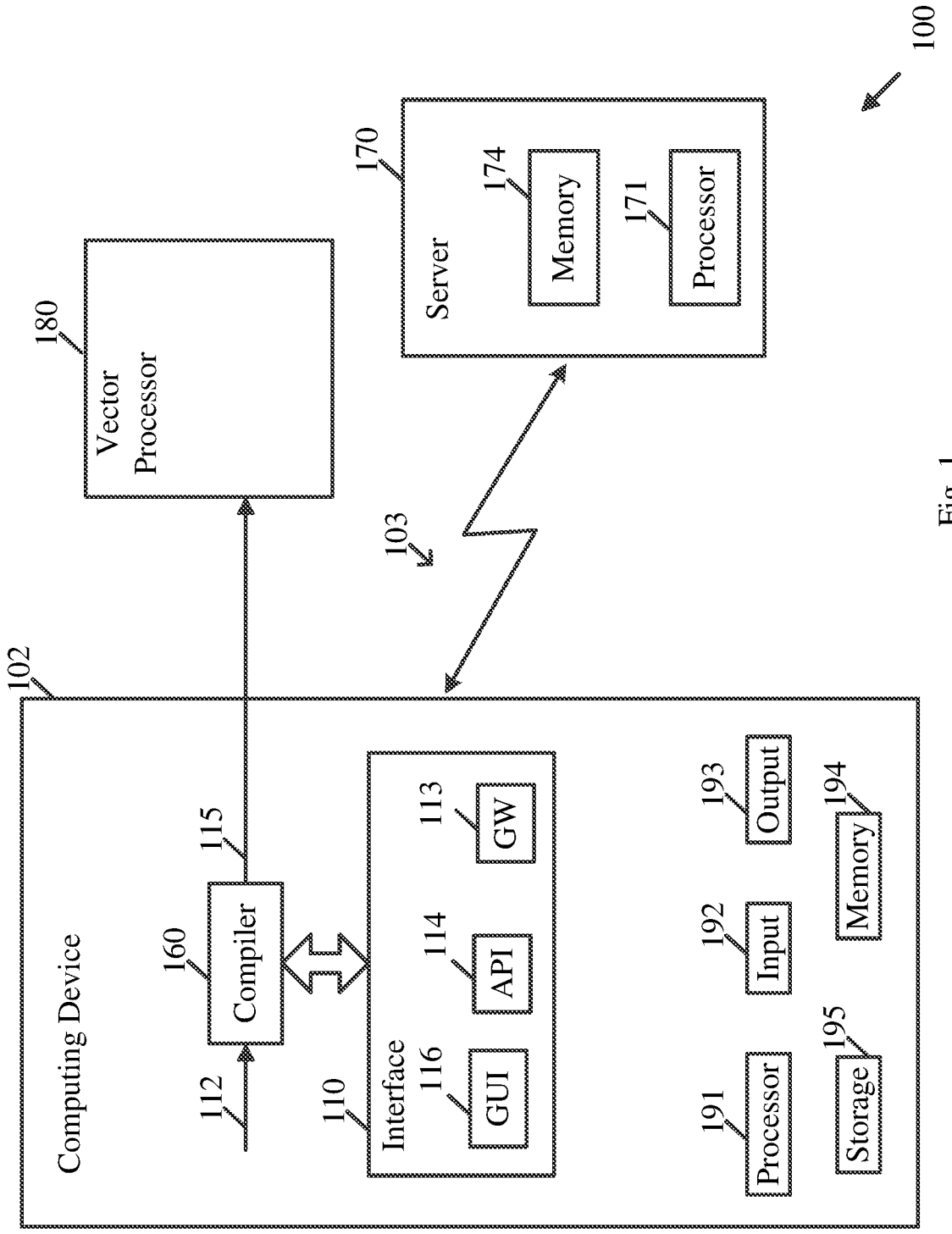
100
102
Computing Device
103
180
Vector Processor
170
Server
174
Memory
171
Processor
160
Compiler
115
112
110
Interface
116
GUI
114
API
113
GW
191
Processor
192
Input
193
Output
194
Memory
195
Storage

Fig. 1

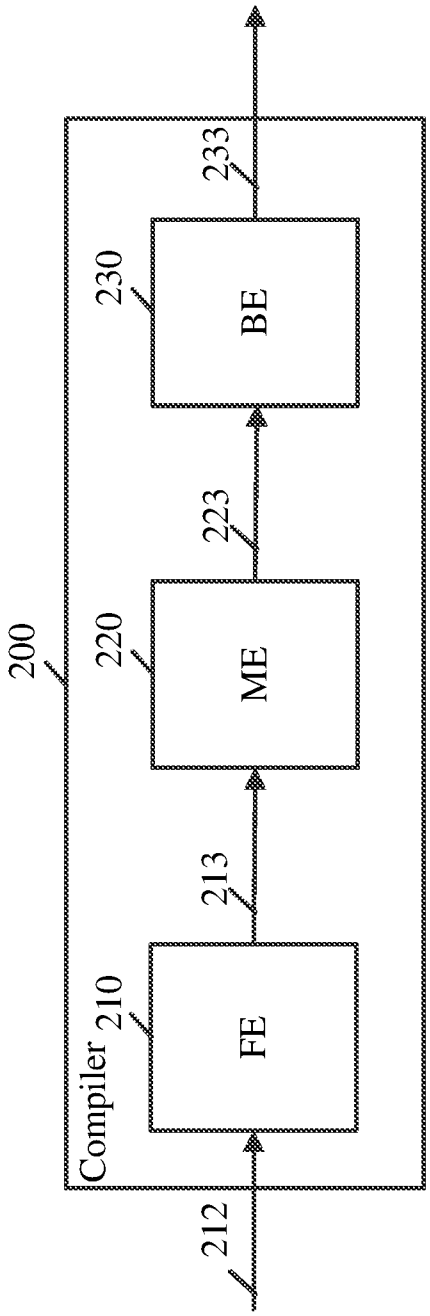

Fig. 2

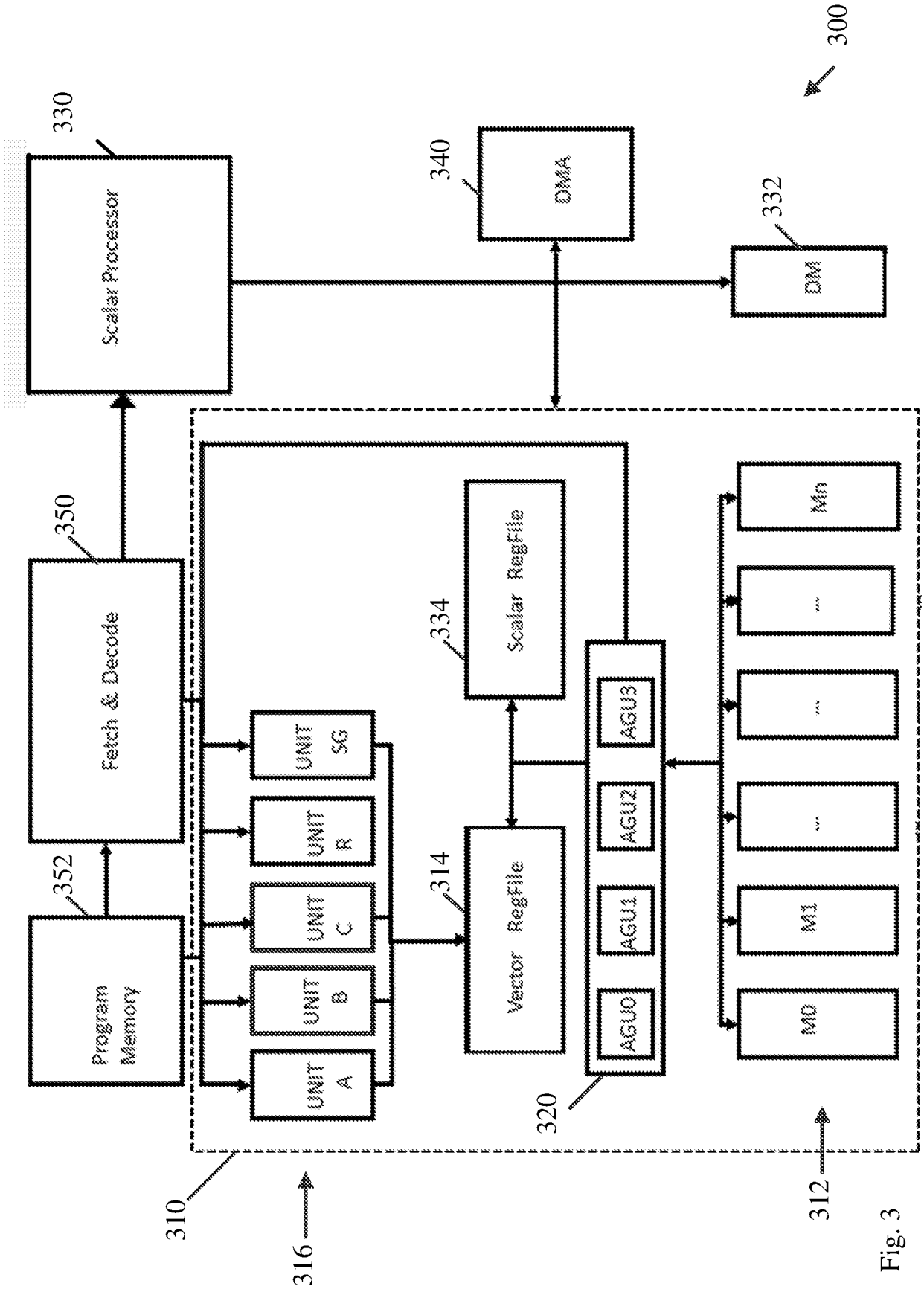
300
330
Scalar Processor
340
DMA
332
DM
350
Fetch & Decode
352
Program Memory
UNIT A
UNIT B
UNIT C
UNIT R
UNIT SG
314
Vector RegFile
334
Scalar RegFile
320
AGU0
AGU1
AGU2
AGU3
M0
M1
...
...
...
Mn
310
316
312

Fig. 3

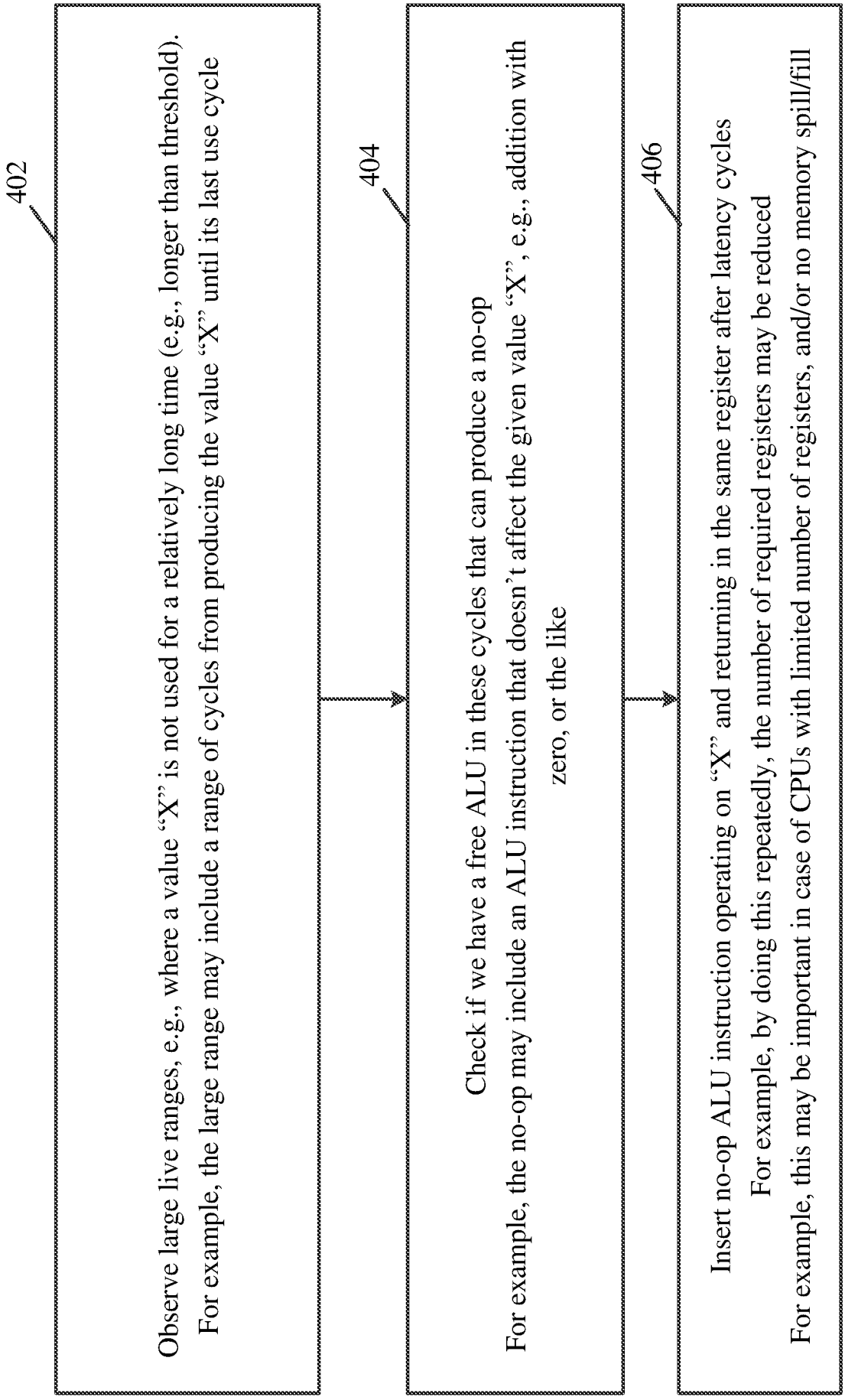
402
Observe large live ranges, e.g., where a value "X" is not used for a relatively long time (e.g., longer than threshold).
For example, the large range may include a range of cycles from producing the value "X" until its last use cycle
404
Check if we have a free ALU in these cycles that can produce a no-op
For example, the no-op may include an ALU instruction that doesn't affect the given value "X", e.g., addition with zero, or the like
406
Insert no-op ALU instruction operating on "X" and returning in the same register after latency cycles
For example, by doing this repeatedly, the number of required registers may be reduced
For example, this may be important in case of CPUs with limited number of registers, and/or no memory spill/fill

Fig. 4

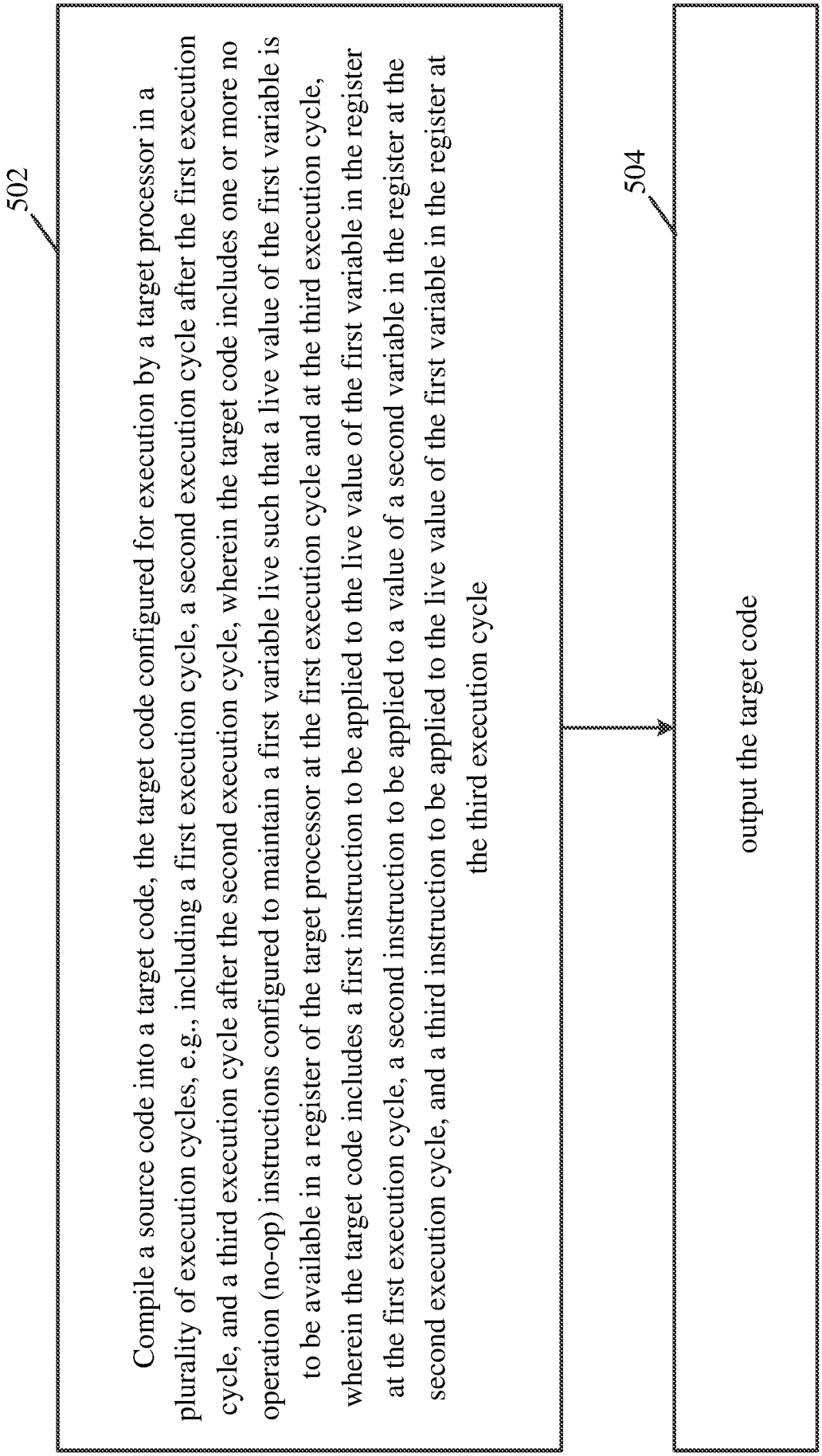
502
Compile a source code into a target code, the target code configured for execution by a target processor in a plurality of execution cycles, e.g., including a first execution cycle, a second execution cycle after the first execution cycle, and a third execution cycle after the second execution cycle, wherein the target code includes one or more no operation (no-op) instructions configured to maintain a first variable live such that a live value of the first variable is to be available in a register of the target processor at the first execution cycle and at the third execution cycle, wherein the target code includes a first instruction to be applied to the live value of the first variable in the register at the first execution cycle, a second instruction to be applied to a value of a second variable in the register at the second execution cycle, and a third instruction to be applied to the live value of the first variable in the register at the third execution cycle
504
output the target code

Fig. 5

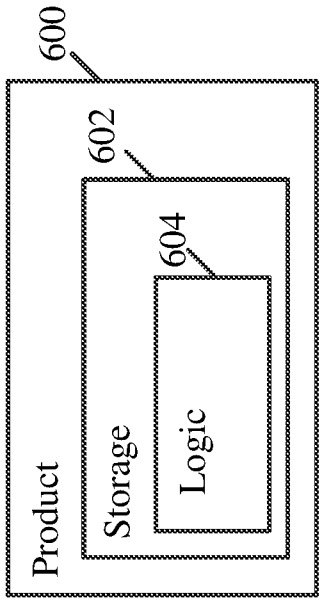

Fig. 6

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB2023/060302

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F8/41
ADD.

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)

EPO-Internal, INSPEC, IBM-TDB, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	IVAN D BAEV ET AL: "Prematerialization", PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES, ACM, 2 PENN PLAZA, SUITE 701 NEW YORK NY 10121-0701 USA, 16 September 2006 (2006-09-16), pages 285-294, XP058335396, DOI: 10.1145/1152154.1152197 ISBN: 978-1-59593-264-8 abstract page 285, right-hand column, paragraph 2 - page 286, left-hand column, paragraph 3 page 287, left-hand column, paragraph 3 - page 290, left-hand column, last paragraph ----- -/--	1-27

[x] Further documents are listed in the continuation of Box C.

[ ] See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance;; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance;; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
2 February 2024	15/02/2024

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Lelait, Sylvain

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB2023/060302

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	STEVEN M KURLANDER ET AL: "Zero-cost range splitting", ACM SIGPLAN 1994 CONFERENCE ON PROGRAMMING LANGUAGE DESIGN AND IMPLEMENTATION (PLDI '94). ORLANDO, FLORIDA, UNITED STATES, JUNE 20 - 24, 1994; [SIGPLAN NOTICES ; 29.1994,6], ASSOC. FOR COMPUTING MACHINERY, NEW YORK, N. Y, 1 June 1994 (1994-06-01), pages 257-265, XP058323083, DOI: 10.1145/178243.178420 ISBN: 978-0-89791-662-2 abstract page 257, left-hand column, paragraph 2 page 258, right-hand column, last paragraph - page 260, left-hand column, paragraph 4 page 261, left-hand column, paragraph 2 -----	1-27