# CUDA内存模型

对于编程语言，内存的访问和管理都是非常重要组成部分。在依靠硬件加速器的高性能计算领域，内存管理对性能会产生更大的影响。

许多应用程序包含大量数据读取和写入操作，这些操作往往会产生大量的延迟。如何快速的进行数据操作，提高数据读写效率，对降低访存延迟，提高内存带宽，获得更高的性能至关重要。

硬件方面，访存延迟低，容量大的内存从技术方面或经济方面往往是不可能的。因此，目前的计算机普遍采用内存层次结构模型来达到低延迟、高容量的目的。内存层次结构设计原理依赖于时间局部性和空间局部性。时间局部性假设如果某一个数据空间被访问，那么在一个短时间窗口内，这个数据空间会被再次访问。空间局部性假设如果某一个数据空间被访问，那么其相邻数据空间有可能同时被访问。内存层次结构由不同等级的存储器组成，图1-1展示了目前典型的内存层次结构，包含寄存器、缓存、内存、硬盘四个等级。存储器等级越高，访存延迟和消耗越低，带宽越高，但价格越贵，容量越小。内存层次结构能够使高时间局部性和空间局部性的数据存储到高等级的存储器中，从而实现大容量、低延迟的目的。

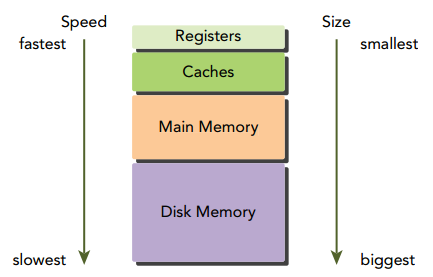


图1-1 典型的内存层次结构

内存层次结构能够从硬件方面提高访存性能，在软件方面，开发人员则可以优化应用程序，减少应用程序中不必要的数据读写操作，软硬结合，从而获得更大的性能提升。为了达到这一要求，开发人员必须充分了解编程语言的内存模型，利用局部性原理高效地对内存进行管理，以获得最低的延迟、最大的带宽和最高的性能。

CUDA作为目前流行高性能平台，有一套特定的内存模型，如图1-2所示。

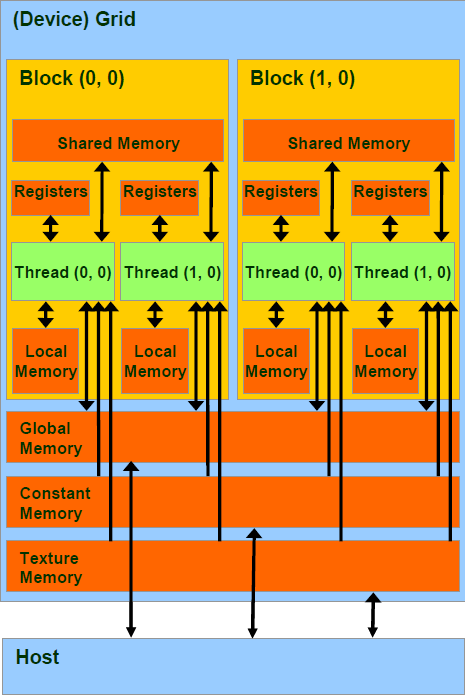


图1-1 CUDA存储器模型

在CUDA的内存模型中，共有六种类型的存储器，分别是寄存器、共享内存、本地内存、常量内存、纹理内存和全局内存。这六种内存可以分为两类，片上内存和片外内存。片上内存位于每一个流多处理器（SM, Stream-Multiprocessor）内，包括寄存器、共享内存和本地内存；片外内存可以被所有流多处理器中的线程访问，包括常量内存、纹理内存和全局内存。在近几代GPU架构（Fermi、Kepler）中，不同内存有不同的性质。以Fermi架构与Kepler架构对比为例，Fermi架构为16个SM ，Kepler架构包含15个SM；寄存器的大小由32KB（Fermi）增长到64KB；L1 cache和共享内存的配置选项由Fermi架构的两个（共享内存-L1 cache：16KB-48KB， 48KB-16KB）增长到三个（16KB-48KB， 48KB-16KB，32KB-32KB）；L2 cache大小由768KB增长到1536KB，全局内存由6GB增长到12GB。

每一种内存都有不同的访存范围、生命周期和功能。核函数中的每一个线程拥有一块私有的本地内存，当线程中局部变量和数组有可能大于寄存器的容量时数据将被放到本地内存中，比如动态数组和大结构体或数组。本地内存的生命周期与线程相同；每一个线程块私有一块共享内存，该共享内存内的数据可以被同一线程块内的所有线程访问，共享内存的生命周期与线程块相同；常量内存和纹理内存都是可以被所有线程访问的只读内存，它们分别针对不同的应用场景进行了优化，纹理内存具有硬件插值、数据类型强制转化的功能，常量内存用于存储常量，具有低访存延迟、广播的功能。全局内存可供所有的线程进行读写操作。全局内存、常量内存和纹理内存内的数据在一个内核函数执行完成后将被继续保持，可以被同一程序中其也内核函数调用，生命周期与应用程序的生命周期相同。

在本章节中，将对常用的三种内存（全局内存、共享内存、常量内存）进行详细介绍。

## 1.1 全局内存

全局内存（Global Memory）是最大的、最常用的GPU内存，同时也是访存延迟最高的内存。Global代表该是指其作用域和使用寿命。在设备上的任何SM中的线程都对全局内存进行读写访问，其生命周期即整个应用程序的生命周期。

全局内存包括静态和动态两种申请方式。开发者可以使用\_\_device\_\_静态申请全局变量，也可以使用cud​​aMalloc()函数动态申请和使用cudaFree()函数动态释放的全局内存​​。全局内存指针可以作为核函数的参数传递到设备端。

线程执行时不能进行跨线程块同步。当多个线程同时写全局内存中相同位置的数据很有可能会造成访存冲突，这将导致不确定的程序行为。这种情况下，必须只用原子操作以保住数据写操作的正确性。

全局内存的访问带宽为32字节，64字节或128字节。当访问全局内存时，数据大小会根据访存带宽被自动对齐，即第一个数据地址必须是32，64或128的倍数。优化全局内存的访问行为将会大大优化应用的性能。当一个warp内的线程对全局内存中的数据进行读取时，所加载的数据大小由两个因素决定：①所涉及数据覆盖的内存大小；②内存地址的对齐。例如一个warp内的所有线程访问的数据大小为60字节，根据内存对齐原则，LDST（Load/Store）单元将读取64字节的数据到缓存中。

## 1.2 共享内存

在核函数中，用\_\_shared\_\_修饰的变量存储在共享内存中。共享内存也包括静态和动态两种申请方式。开发者这在核函数中使用\_\_shared\_\_ int sharedData[256]语句为每一个线程块静态地分配一块256\*sizeof(int)大小的共享内存；核函数在调用时，共有四个可配置参数，格式如下：

|  |
| --- |
| kernel\_name <<<grid\_size, block\_size, shared\_memory\_size, stream>>>(argument list); |

其中，grid\_size为核函数中block的数量，block\_size为一个block内的线程数量，shared\_memory\_size为动态申请的共享内存大小单位为byte，stream表示该核函数处在哪个流中执行，初始值为零，表示在默认流中执行。共享内存的动态申请方式为在核函数中不指定所使用共享内存大小，语句为：extern \_\_shared\_\_ int sharedData[]，由核函数的第三个参数动态设置线程块中可用共享内存大小。

共享内存为片上内存，相比于全局内存，共享内存具有更高的带宽和更低的访存延迟。然而每一个SM具有的共享内存大小有限，Kepler架构每个SM的默认共享内存大小为48K。众所周知，SM中可同时运行的warp数量由两个个因素决定：共享内存资源和寄存器资源。所以开发者必须谨慎地使用共享内存，在线程块中过量的使用共享内存将会减少SM可承载的warp数量，降低核心占有率从而导致性能下降。共享内存在内核函数的内声明，其生命周期与线程块相同，当一个线程块执行完毕，它所占有的共享内存将被释放并分配给其他线程块。

共享内存是线程块内线程间通信的基本手段。线程块中的线程可以通过共享内存的数据操作来进行通信。使用共享内存必须使用\_\_syncthreads（）函数进行同步。该函数创建一个屏障，同一个线程块中的所有线程必须全都到达屏障，线程才被允许继续执行。屏障的作用在于可以防止潜在的数据风险，比如脏数据，读写冲突等。

在每个SM中，共享内存和L1缓存共用同一块片上内存，它是静态地分配，但可在使用如下函数在运行时动态配置的：

|  |
| --- |
| CUDAError\_t CUDAFuncSetCacheConfig(const void\* func, enum CUDAFuncCache cacheConfig); |

该函数配置基于每个内核基础片上存储器的分区，设置用于通过func指定的内核函数的配置。以Kepler架构为例，支持的高速缓存配置如下：

|  |
| --- |
| CUDAFuncCachePreferNone：没有偏好（默认）  CUDAFuncCachePreferShared：48KB共享内存和16KB L1缓存  CUDAFuncCachePreferL1：48KB L1高速缓存和16KB共享内存  CUDAFuncCachePreferEqual：L1缓存和共享内存的大小相等分别为32KB |

## 1.3 常量内存

常量内存作为一块独特的内存区域，每个SM都有一个专用的常量缓存。用\_\_constant\_\_修饰的变量存储在常量内存中。常量内存必须在核函数之外的全局作用域进行声明。常量内存大小有限，在所有计算能力的显卡中，最多可使用 64 KB的常量内存。常量内存为只读内存，可以被同一个GPU的所有核函数中的线程进行读访问。

常量内存必须静态声明，在声明时必须指定数据类型和大小，如\_\_constant\_\_ int constant\_data[256]。常量内存赋值数据有两种方式，第一种是在声明时直接初始化赋值，第二种方式为在host端生成数据，调用cudaMemcpyToSymbol()函数将host端数据复制到常量内存中。

常量内存的使用场景较为特殊，当一个warp的所有线程从同一个内存地址读取数据情况时，常量内存会将该数据广播到该warp的其他的线程中，这种使用方式常量内存的性能最佳。例如，对于一个数学公式而言，将系数存储在常量内存中将获得最佳性能。

常量内存每一个时钟周期内的最大吞吐量为4B, 如果一个warp内的所有线程所需数据大小大于4B, 则需要多次读取。假如一个warp中的每个线程读取常量内存中不同数据，这种情况下，常量内存是不是最好的选择。

# 2 测试过程设计

## 2.1 需求分析

本测试报告的内容为测试GPU不同内存访问模式下的性能，需要考虑的因素如下：

1.不同形式的数据组织形式

2.不同的访问方式

3.不同的内存

4.不同的线程数目

经过分析，该实验这个实验需要考虑的因素包括：数据存储方式（一维数组、二维数组等），数据拷贝方式（直接拷贝、对齐拷贝、数组拷贝等），数据存储位置（全局内存、共享内存、常量内存），数据访问方式（顺序访问、step访问、访问次数）等等因素。

不同的情景下，最佳的访问方式可能截然不同。实验设计者需要对当前CUDA内常用的访问方式比较了解，才能较有意义地设计本次实验。

## 2.2 实验设计

根据需求分析，从数据组织形式、数据内部分布、数据访问方式和数据存储位置四个因素对该测试实验进行详细设计。

### 2.2.1 数据组织形式

本实验中选取三种常见的数据组织形式：一维数组、二维数组和树。详细数据结构如下：

**一维数组**

数据结构：DATA\_TYPE \*data;

**二维等长数组**

因为GPU内存拷贝形式的特殊性，在这里用一维模拟二维数组

数据结构：

|  |
| --- |
| struct **{**  size\_t width**;**  size\_t height**;**  DATA\_TYPE **\***imgData**;**  size\_t pitchBytes**;**  **}** |

**树结构**

树的表示方式有两种，链表存储方式和顺序存储方式，显然在GPU中链表存储方式是不可取的，会造成大量的不友好的数据读写操作。本设计中选取顺序存储方式为树的表示方式。

顺序存储方式有三种：双亲表示法,、孩子表示法,、孩子兄弟表示法。

综合考虑，本测试文档中的树结构统一使用完全二叉树，存储方式选用孩子表示法

数据结构：

|  |
| --- |
| **typedef** struct Node\_st**{**  DATA\_TYPE data**;**  Node**\*** left**;**  Node**\*** right**;**  **}**Node**;**  **typedef** struct  **{**  Node nodes**[**MAXSIZE**];**  int r**,**n**;** //根结点的位置和结点数  **}**Tree**;** |

### 2.2.2 数据内部分布

数据内部分布即数组内数据的分布特点，本测试实验选取了常用的六种分布，分布及分布公式如下：

（1）随机分布

（2）正态分布

http://h.hiphotos.baidu.com/baike/s%3D205/sign=2abf505a42166d223c77129473220945/342ac65c1038534384b650b09213b07eca808822.jpg

（3）泊松分布

http://c.hiphotos.baidu.com/baike/s%3D114/sign=bc59b0147dd98d1072d40830153eb807/574e9258d109b3de0e06c280cdbf6c81810a4c98.jpg

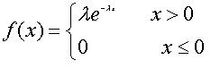
（4）均匀分布

Y = x

（5）几何分布

http://g.hiphotos.baidu.com/baike/s%3D219/sign=990b72acb68f8c54e7d3c22e03282dee/314e251f95cad1c891dc48657d3e6709c93d5108.jpg

（6）指数分布



### 2.2.3 数据访问方式

在核函数中，线程根据下标访问数据，数据访问方式即数据访问下标的规律。本测试实验选取了常用的七种分布作为数据访问方式，分别是顺序访问，Step访问，随机访问，标准正态分布访问，泊松分布访问，几何分布访问，和指数分布访问。

### 2.2.4 数据存储位置

选取常用的三种内存，全局内存，常量内存和共享内存。

### 2.2.5 其他

数据访问方式汇总，step访问中step取2和4。

线程块大小取：256,512和1024。

数据元素个数，

全局内存取：1024, 4096, 10240, 40960, 102400

共享内存取：512, 1024, 4096, 10240

常量内存取：512,1024,10240,40960

数据类型取：unsigned char

注意事项：因为sizeof(Node)=24，所以树结构情况下，常量内存数据元素个数取：128，512，1024,2048。

## 2.3 测试代码架构

### 2.3.1 代码目录结构

本项目共有8个文件，三大部分（Case，Common，Distribution）、一个main文件和一个Makefile文件。

Common.h/cu：文件中包括全局变量、枚举变量、结构体和基础类的定义。

Distribution.h/cu：文件中包括生成各种分布数据生成函数的声明和定义。

Case.h/cu：核心逻辑类。

Main.cu：主函数。

### 2.3.2 Common.h/cu

**（1）全局变量**

全局变量包括：测试数据类型、测试数据范围（MIN、MAX）、PI以及e。

其中测试数据范围是与测试数据类型有关。

|  |
| --- |
| // 测试数据的类型  #define DATA\_TYPE unsigned char  // 最大最小值跟 DATA\_TYPE 的类型有关  #define MIN 0  #define MAX 256  #ifndef PI  #define PI 3.141592654  #endif  #ifndef e  #define e 2.718281828459  #endif |

**（2）枚举变量**

定义了数据内容分布类型、数据访问类型和数据组织形式。

其中数据内容分布类型有六种，数据访问类型有七种，数据组织形式有三种。详见注释。

|  |
| --- |
| // 数据内部分布  enum data\_content  **{**  dc\_random **=** 0**,** // 随机分布  dc\_standard\_normal**,** // 标准正态分布  dc\_poisson**,** // 泊松分布  dc\_uniform**,** // 均匀分布  dc\_geometric**,** // 几何分布  dc\_exponential // 指数分布  **};**  enum access\_mode  **{**  am\_sequential**=**0**,** // 顺序访问  am\_step**,** // step 访问  am\_random**,** // 随机访问  am\_standard\_normal**,** // 标准正态分布访问  am\_poisson**,** // 泊松分布访问  am\_geometric**,** // 几何分布访问  am\_exponential // 指数分布访问  **};**  // 数据组织形式  enum data\_form  **{**  df\_1D **=** 0**,**  df\_2D**,**  df\_tree  **};** |

（3）结构体和基础类

包括一维模拟二维的结构体、树节点和树类。

其中Tree结构使用完全二叉树，存储方式选用孩子表示法。

|  |
| --- |
| /\* 树节点 \*/  class Node  **{**  public**:**  Node**\*** left**;**  Node**\*** right**;**  DATA\_TYPE data**;**  **};**  // 一维模拟二维  **typedef** struct data2D\_st  **{**  size\_t rows**;**  size\_t cols**;**  DATA\_TYPE **\***data**;**  size\_t pitchBytes**;**  **}**Data2D**;**  class Tree  **{**  public**:**  // 申请空间并设置父节点和子节点的关系  Tree**(**int \_num**);**  Tree**(){}**  **~**Tree**();**  Node **\***nodes**;**  int num**;** // 节点数量  **};** |

### 2.3.3 Distribution.h/cu

根据分布类型生成各种分布数据和根据。根据数据访问类型生成访问下标数据。

在Distribution.h文件中，对每一种分布重载了三次，分别用于生成一维（或一维模拟二维）数据（DATA\_TYPE\* data）、访问下标数据（int\* data）和树节点数据（Node \*nodes）。实现方式为传出min、max，分布参数、size和数据指针，在函数内生成数据并赋值。

|  |
| --- |
| int random**(**int min**,** int max**,** int size**,** DATA\_TYPE**\*** data**);**  int random**(**int min**,** int max**,** int**\*** data**);**  int random**(**int min**,** int max**,** int size**,** Node **\***nodes**);**  int standard\_normal**(**int min**,** int max**,** int miu**,** int sigma**,** int size**,** DATA\_TYPE**\*** data**);**  int standard\_normal**(**int min**,** int max**,** int miu**,** int sigma**,** int**\*** data**);**  int standard\_normal**(**int min**,** int max**,** int miu**,** int sigma**,** int size**,** Node **\***nodes**);**  int poisson**(**int min**,** int max**,** int Lambda**,** int size**,** DATA\_TYPE**\*** data**);**  int poisson**(**int min**,** int max**,** int Lambda**,** int**\*** data**);**  int poisson**(**int min**,** int max**,** int Lambda**,** int size**,** Node **\***nodes**);**  int uniform**(**int min**,** int max**,** int size**,** DATA\_TYPE**\*** data**);**  int uniform**(**int min**,** int max**,** int**\*** data**);**  int uniform**(**int min**,** int max**,** int size**,** Node **\***nodes**);**  int geometric**(**double probability**,** int min**,** int max**,** int size**,** DATA\_TYPE**\*** data**);**  int geometric**(**double probability**,** int min**,** int max**,** int**\*** data**);**  int geometric**(**double probability**,** int min**,** int max**,** int size**,** Node **\***nodes**);**  int exponential**(**double lambda**,** int min**,** int max**,** int size**,** DATA\_TYPE**\*** data**);**  int exponential**(**double lambda**,** int min**,** int max**,** int**\*** data**);**  int exponential**(**double lambda**,** int min**,** int max**,** int size**,** Node **\***nodes**);** |

### 2.3.4 Case.h/cu

该类为核心组织类，Case.h内为Case类，代码如下：

|  |
| --- |
| class Case  **{**  public**:**  // 初始化默认参数  Case**()**;  **~**Case**();**  // 数据组织形式  // 一维数组，树等  data\_form df**;**  // 数据大小  int size**;**  int r**,**c**;** // 二维数组所需数据  // 数据内容形式  // 随机、各种分布  data\_content dc**;**  // 根据数据组织形式、数据大小和数据内容形式初始化数据  int initData**();**  // 线程数目  int thread\_num**;**  int block\_size**;** // 线程块大小  // 不同的访问方式  access\_mode am**;**  int step**;**  int am\_num**;** // 访问数据的数量  // 三个运行函数，即核心函数  // 分别得出在不同内存下的访问性能  // 每个函数内执行流程如下：  // (1)（申请空间，初始化数据需提前完成）  // (2)数据拷贝，执行核函数  // (3)释放数据  int global\_run**();**  int shared\_run**();**  int constant\_run**();**  private**:**  // 数据组织形式  DATA\_TYPE **\***data1D**;**  Data2D data2D**;**  Tree **\***tree**;**  **};** |

下面解释类成员变量。

|  |  |
| --- | --- |
| data\_form df; | 数据组织形式，一维、二维和树结构 |
| size | 当为一维数组和树结构时，该成员函数代表测试数据大小 |
| r,c | 当为二维数组时，r代表行数，c代表列数。Size = r\*c |
| data\_content dc; | 数据内容分布 |
| access\_mode am; | 不同分布的访问方式 |
| thread\_num | 线程数量，在这里，线程数量等于数据大小 |
| block\_size | 线程块大小 |
| step | Step访问时所需变量 |
| am\_num | 每个线程访问的数据数目 |

下面解释类成员函数

|  |  |
| --- | --- |
| Case() | 构造函数，初始化各种默认参数 |
| initData(); | 根据df（数据组织形式）和dc（数据内容形式），调用对应的分布函数生成分布数据。 |
| global\_run();  shared\_run();  constant\_run(); | 三个运行核心函数，分别得出在不同内存下的访问性能。函数内流程基本相同。  （1）根据数据组织形式（一维、二维、树结构），申请device空间并拷贝数据；  （2）根据不同的访问分布方式（分为三类，sequential访问、step访问、其他分布访问），执行不同的核函数。  在其他分布访问方式时，需要在核函数外根据不同的访问方式在host端生成访问下标，再拷贝到device端（虽然可以用curand在核函数外生成一些分布，但am\_geometric分布和am\_exponential分布无法生成，为了考虑性能测试统一，不使用curand）。 |

Case.cu内为constant memory定义，核函数和Case类成员函数实现。

Constant memory的大小必须在编译前确定，所以对于不同constant memory数据大小的测试必须每次手动更改。

三种内存（global、shared、constant），三种数据组织形式（一维、二维、树结构），三种访问分布（sequential、step、Common），共27个核函数。核函数名的格式为：\_数据组织形式\_内存种类\_访问分布Ker().

Global memory核函数内流程为：（1）计算下标；（2）下标越界判断（3）读取am\_num个数据，并相加。

Shared memory核函数内流程为：（1）计算下标；（2）下标越界判断；（3）根据copy\_num\_per\_thread拷贝数据到shared memory中。copy\_num\_per\_thread是参数，事先在核函数外计算。（4）读取am\_num个数据，并相加。

Constant memory核函数内流程与global memory核函数相同。

# 3.实验结果

## 3.1 实验环境

硬件环境

|  |  |
| --- | --- |
| 项目 | 描述 |
| CPU | Intel Core i7 2600 |
| 内存 | 12GB DDR3 1333MHz |
| 硬盘 | WD 绿盘 1TB |
| 显卡 | Tesla K40c |

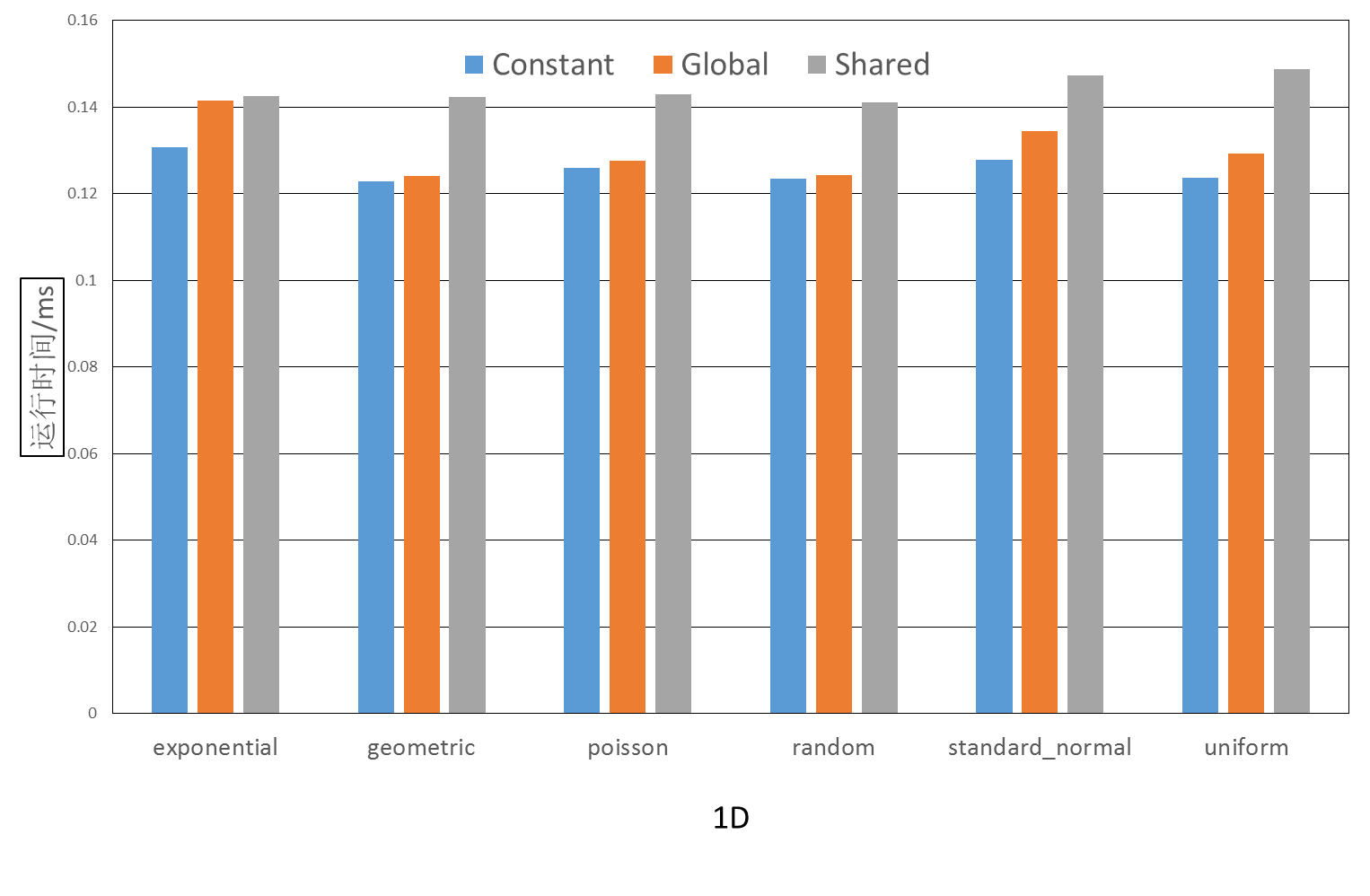
软件环境

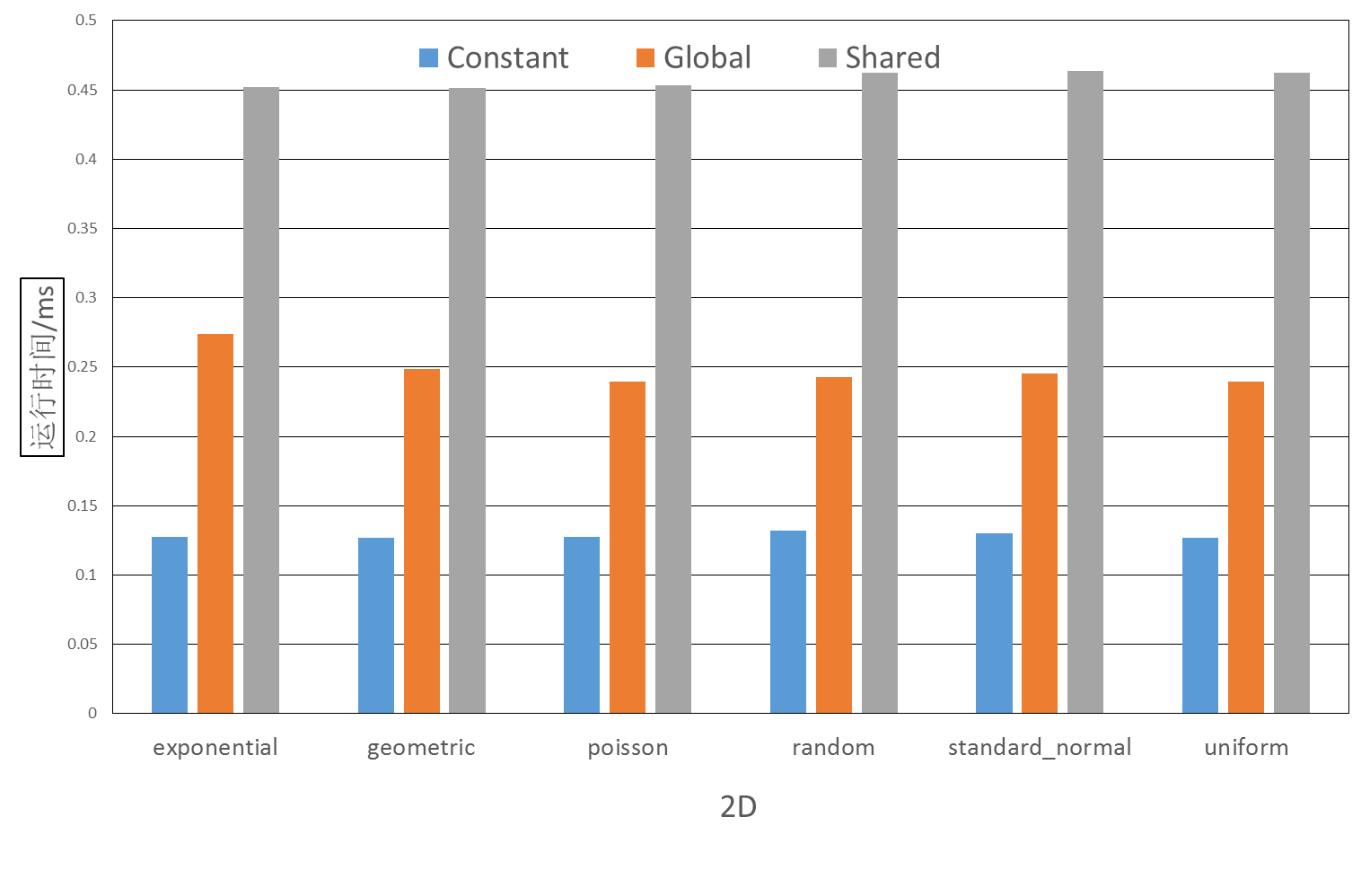
|  |  |
| --- | --- |
| 项目 | 描述 |
| 操作系统1 | Ubuntu 14.04 Desktop x64 |
| CUDA Version | 7.0 |

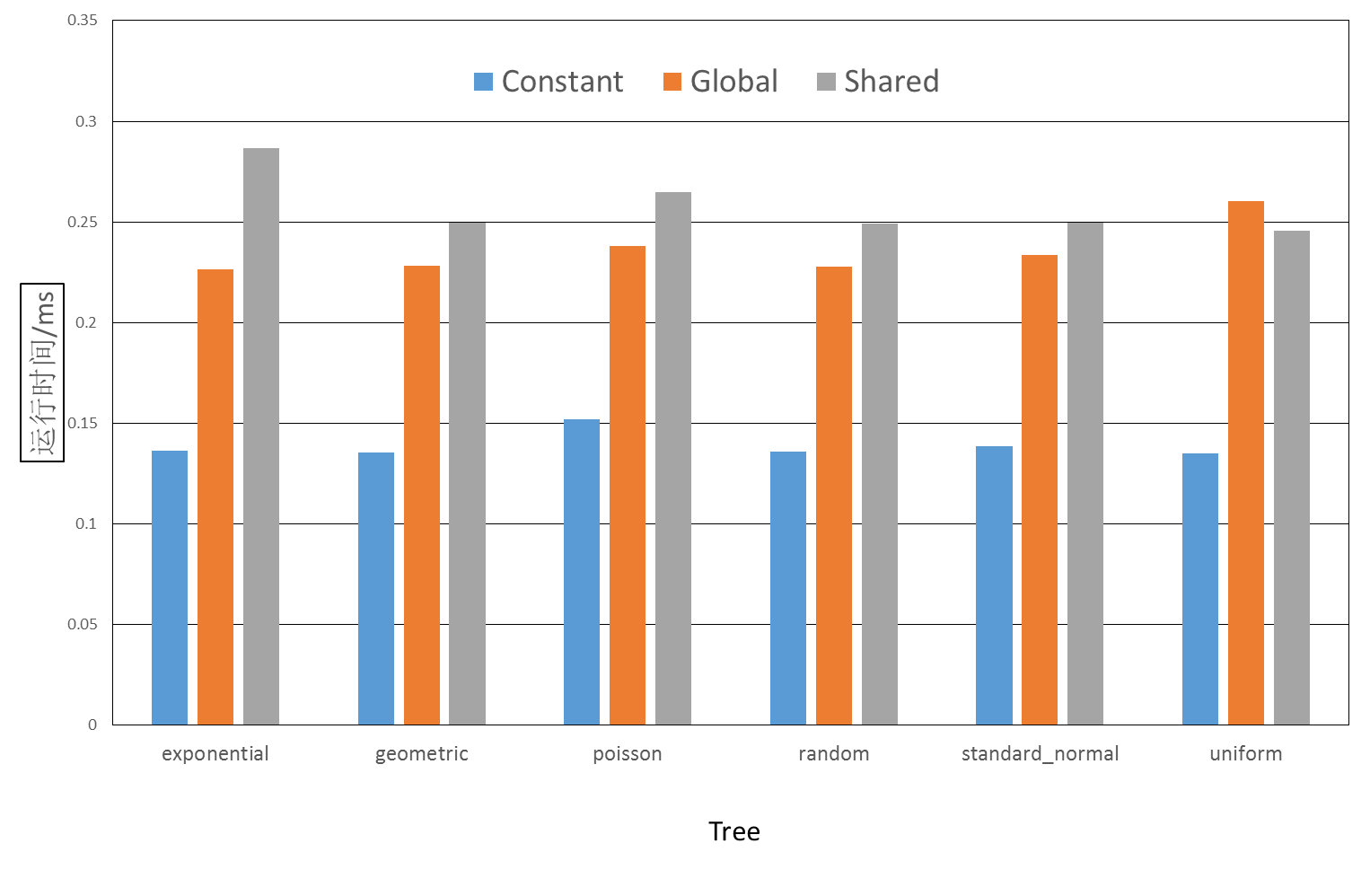
## 3.2 实验分析

### 3.2.1 Best memory

为了探究内存存储位置对数据访问性能的影响，我们选择合理的线程块大小和数据大小，数据访问方式选择默认设置-顺序访问，观察在不同数据内部分布和不同数据组织形式下，哪种内存能够获得最佳性能。实验结果如图\*\*，其中横坐标为数据内部分布，纵坐标代表运行时间。



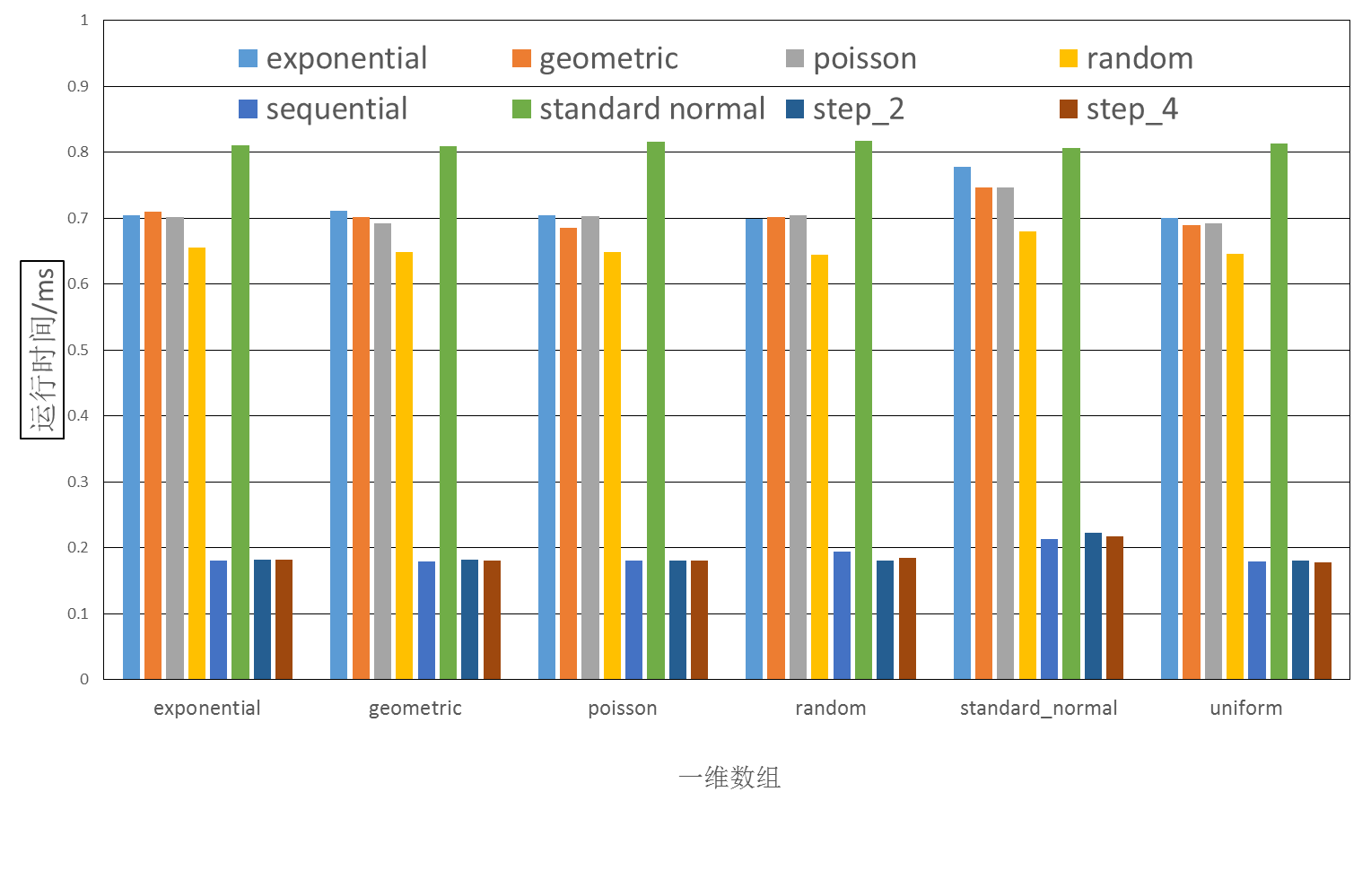


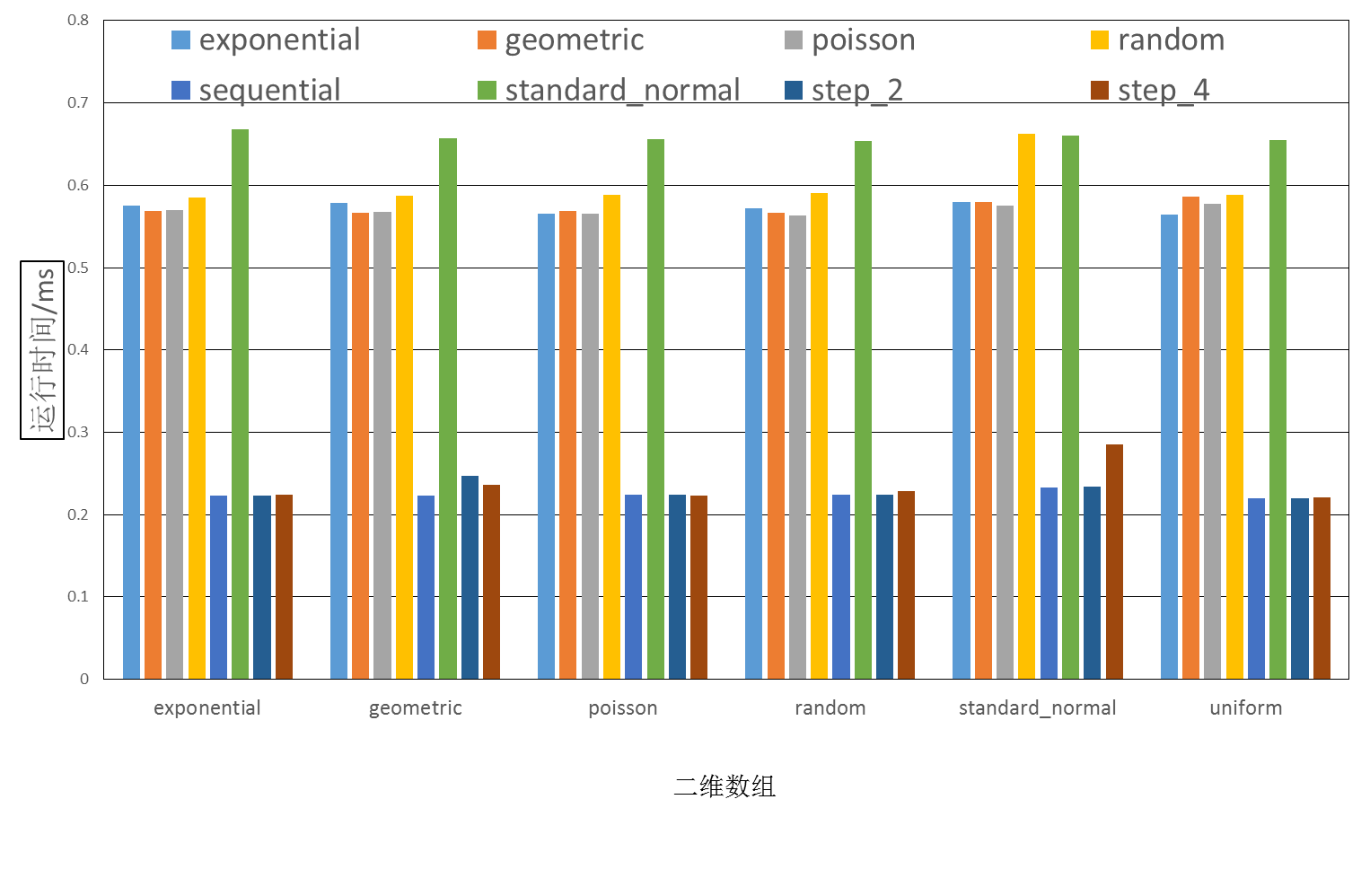


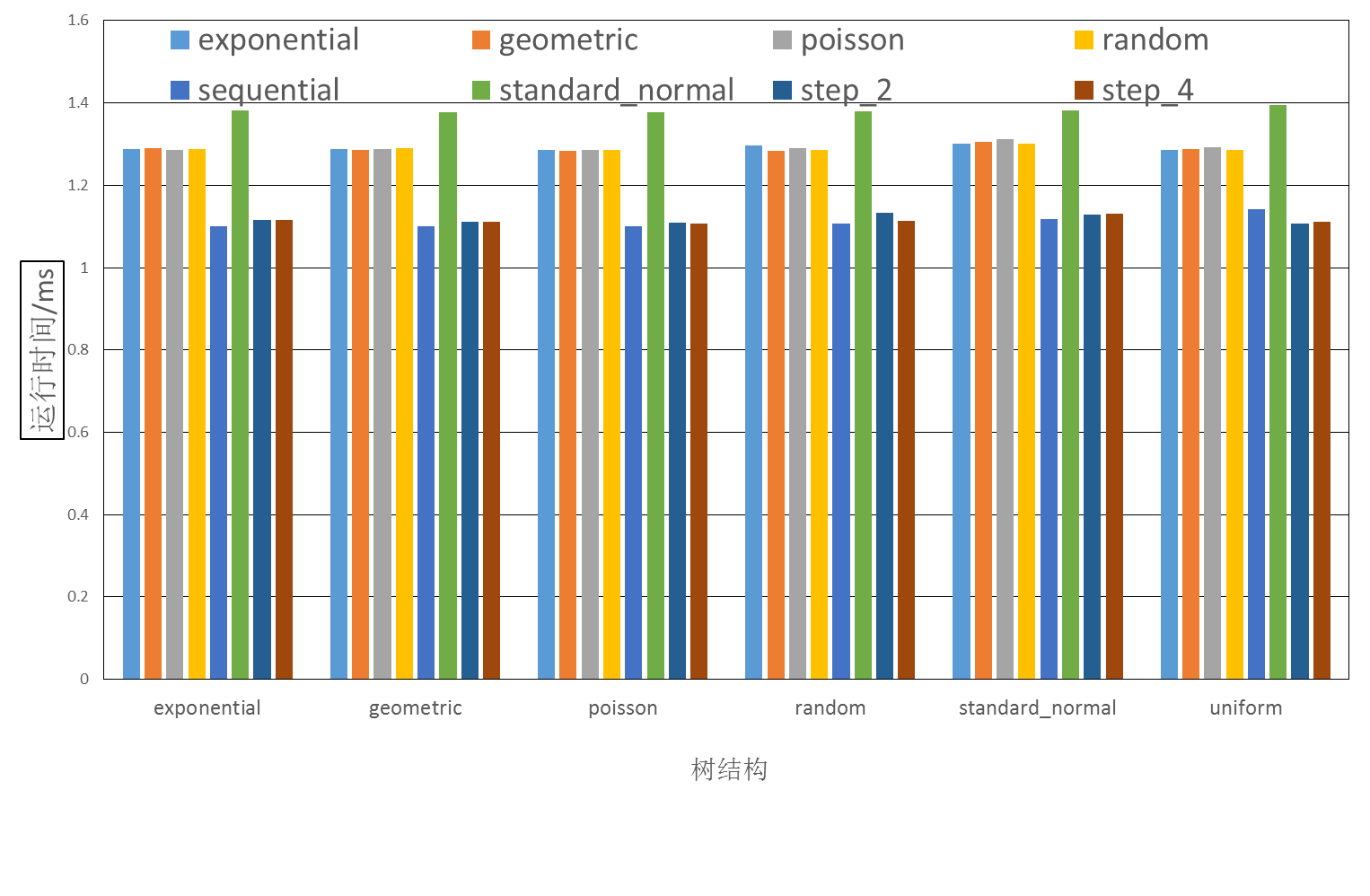
从图中可以看出，在顺序访问方式下，共享内存的性能最差，原因在于每个线程中，线程块中线程从共享内存中读取数据需要先将数据拷贝到共享内存中，这一过程消耗了大量时间。常量内存获得了最好的性能，但常量内存大小有限，数据量较大时，全局内存是最佳选择。

### 3.2.2 Best access mode

为了探究数据访问方式对数据访问性能的影响，我们选择合理的线程块大小和数据大小，内存存储位置选取全局内存，观察在不同数据内部分布和不同数据组织形式下，哪种数据访问方式能够获得最佳性能。实验结果如图\*\*，其中横坐标为数据内部分布，纵坐标代表运行时间。



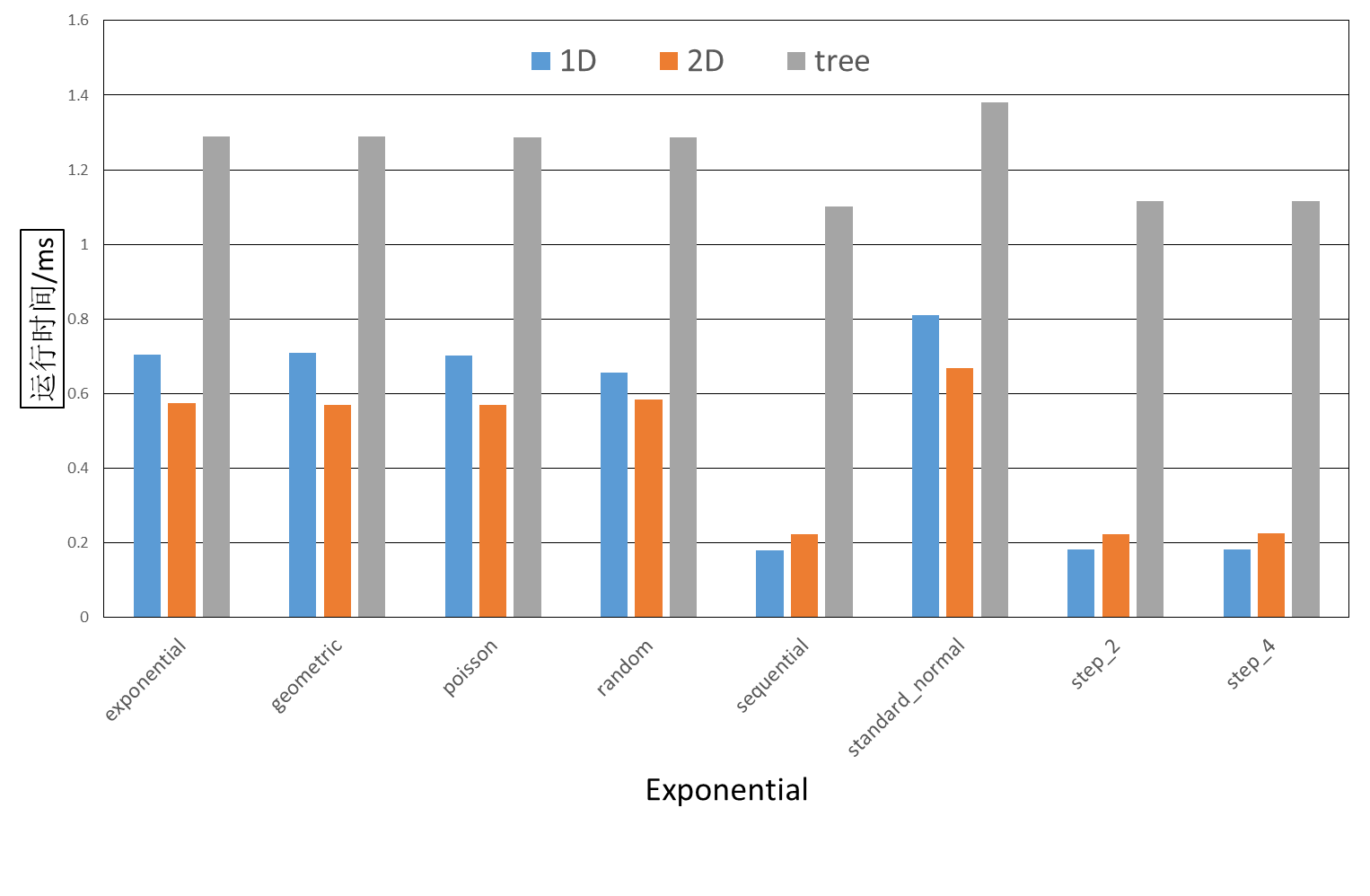


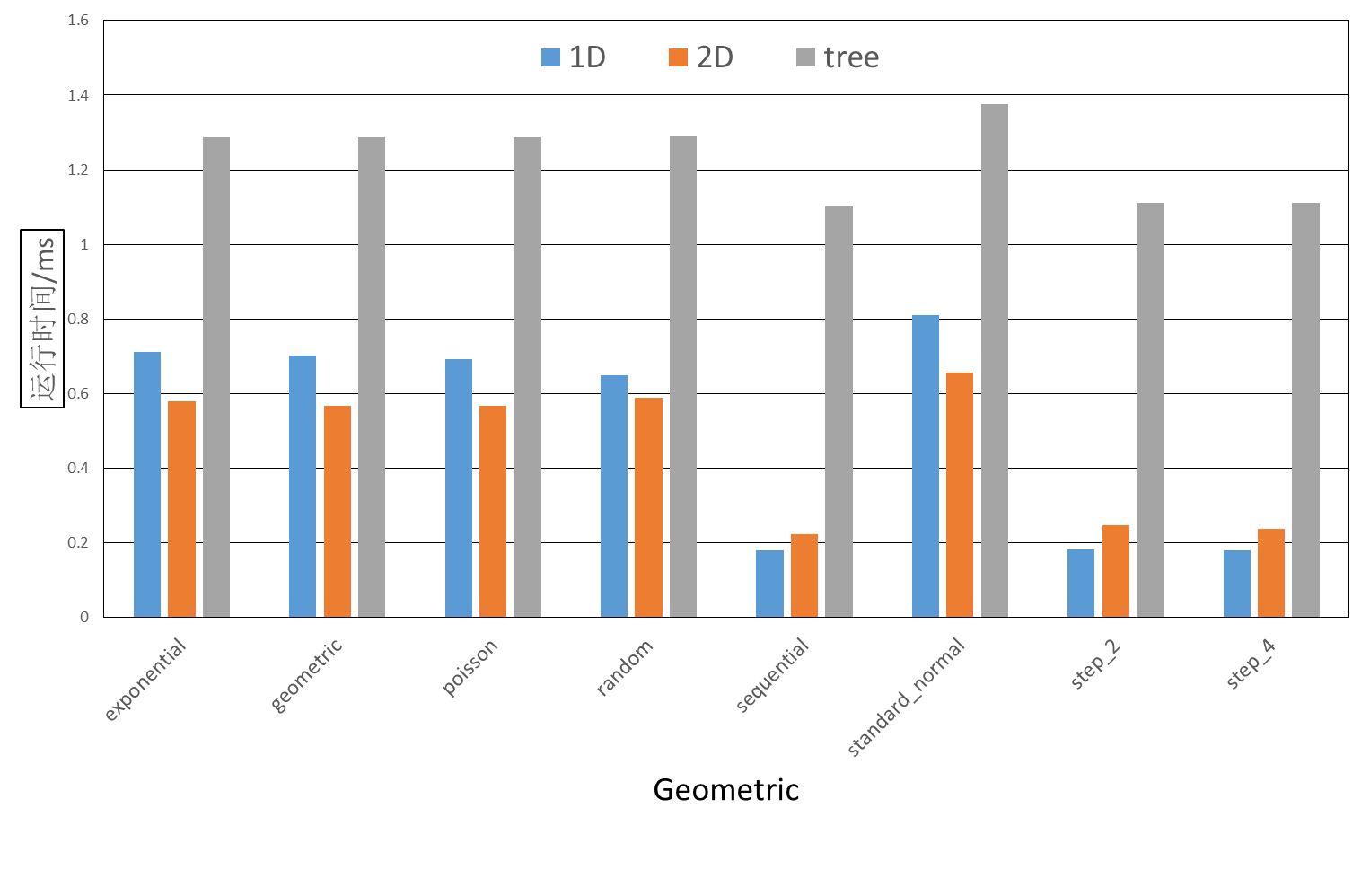


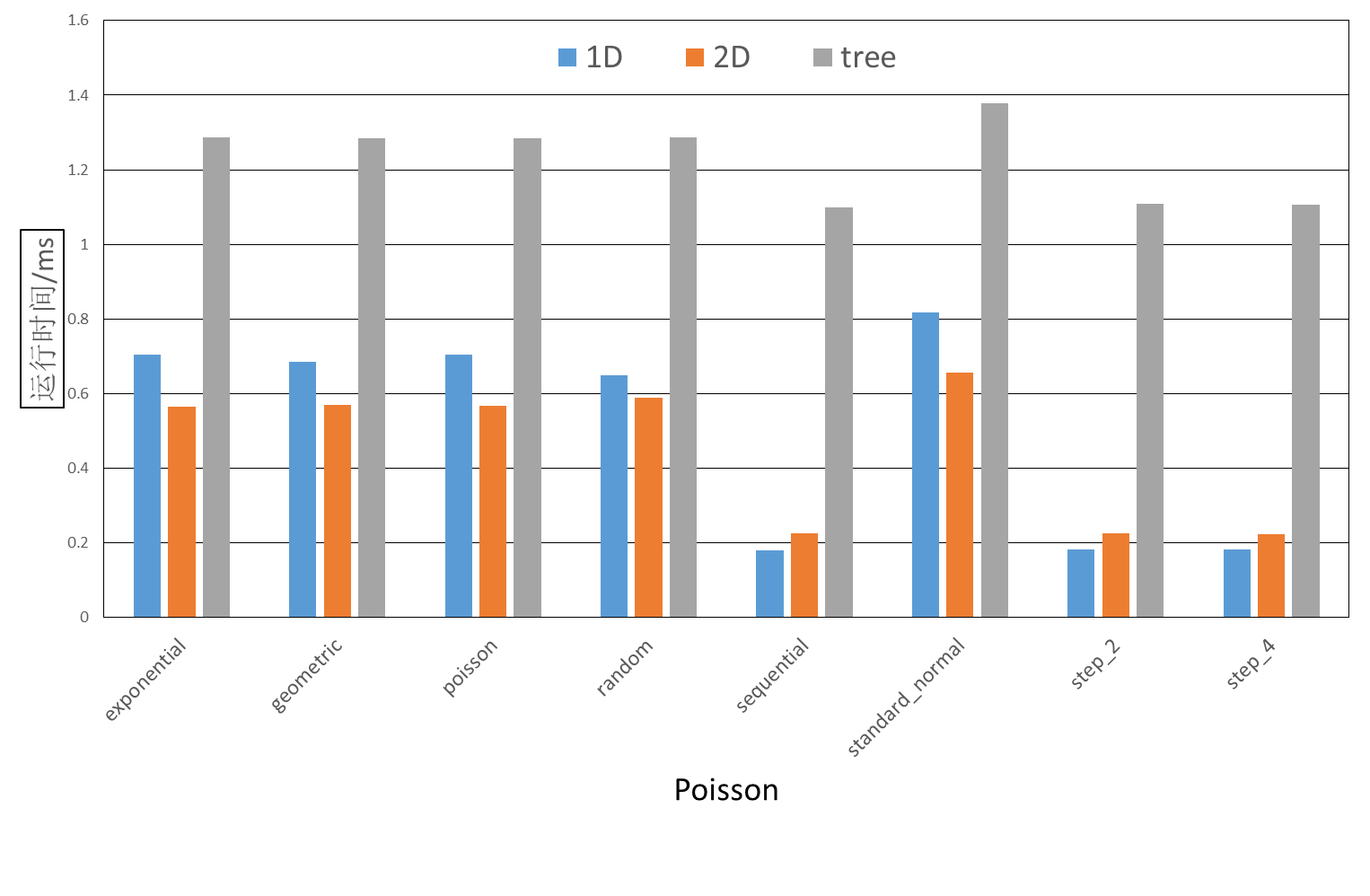
从图中可以看出，对于不同的访问方式，顺序访问（sequential）和step访问（step大小为2、4）的性能基本相同。因为一个L1缓存的缓存行大小是128B，顺序访问方式对cache最友好，step访问次之，因此顺序访问方式性能最好。随机访问、泊松分布访问、几何分布访问和指数分布访问性能大体相同。而标准正态分布产生的访问序列跨度较大，对cache最不友好，性能最差。

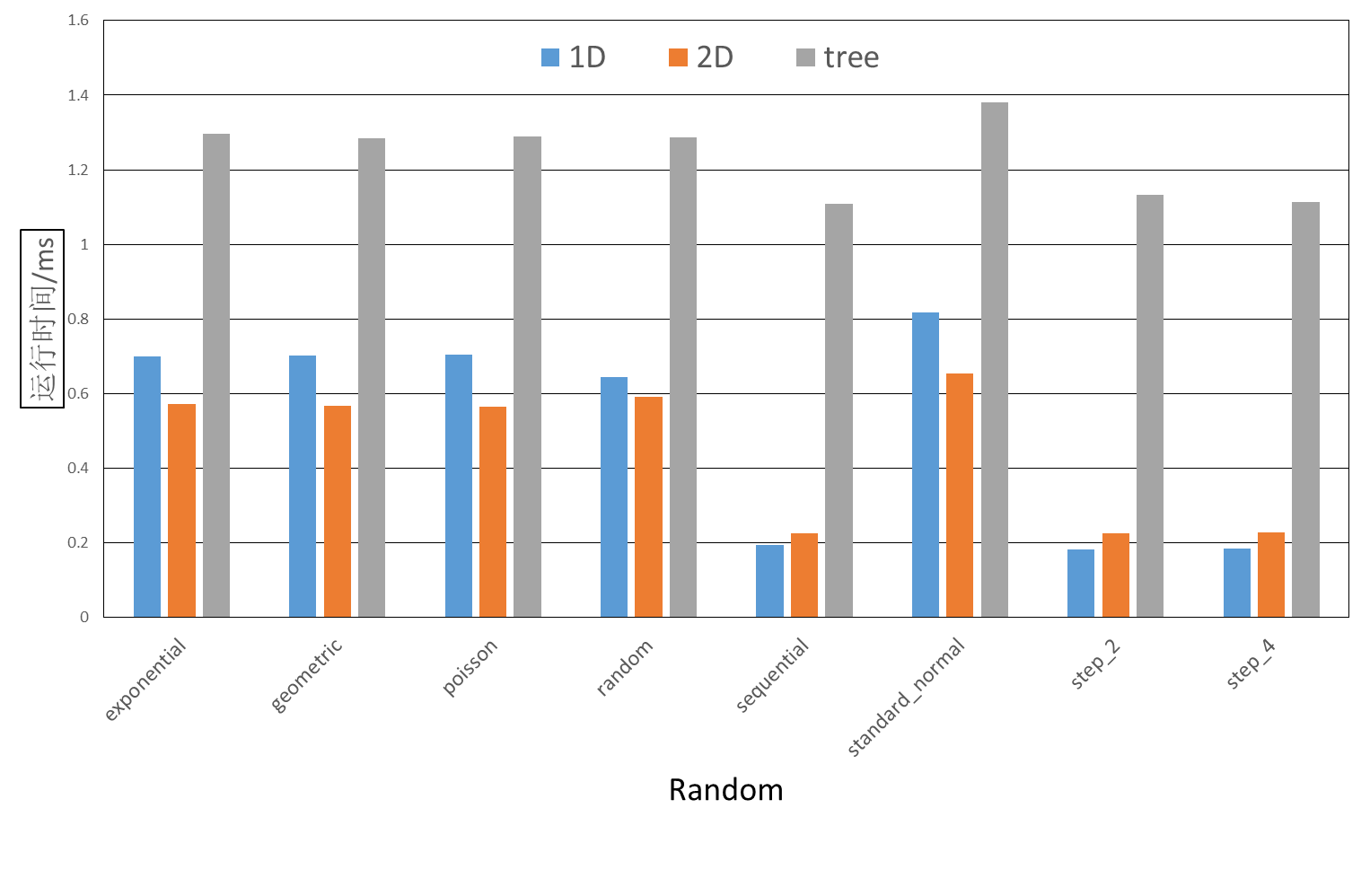
### 3.2.3 Best data form

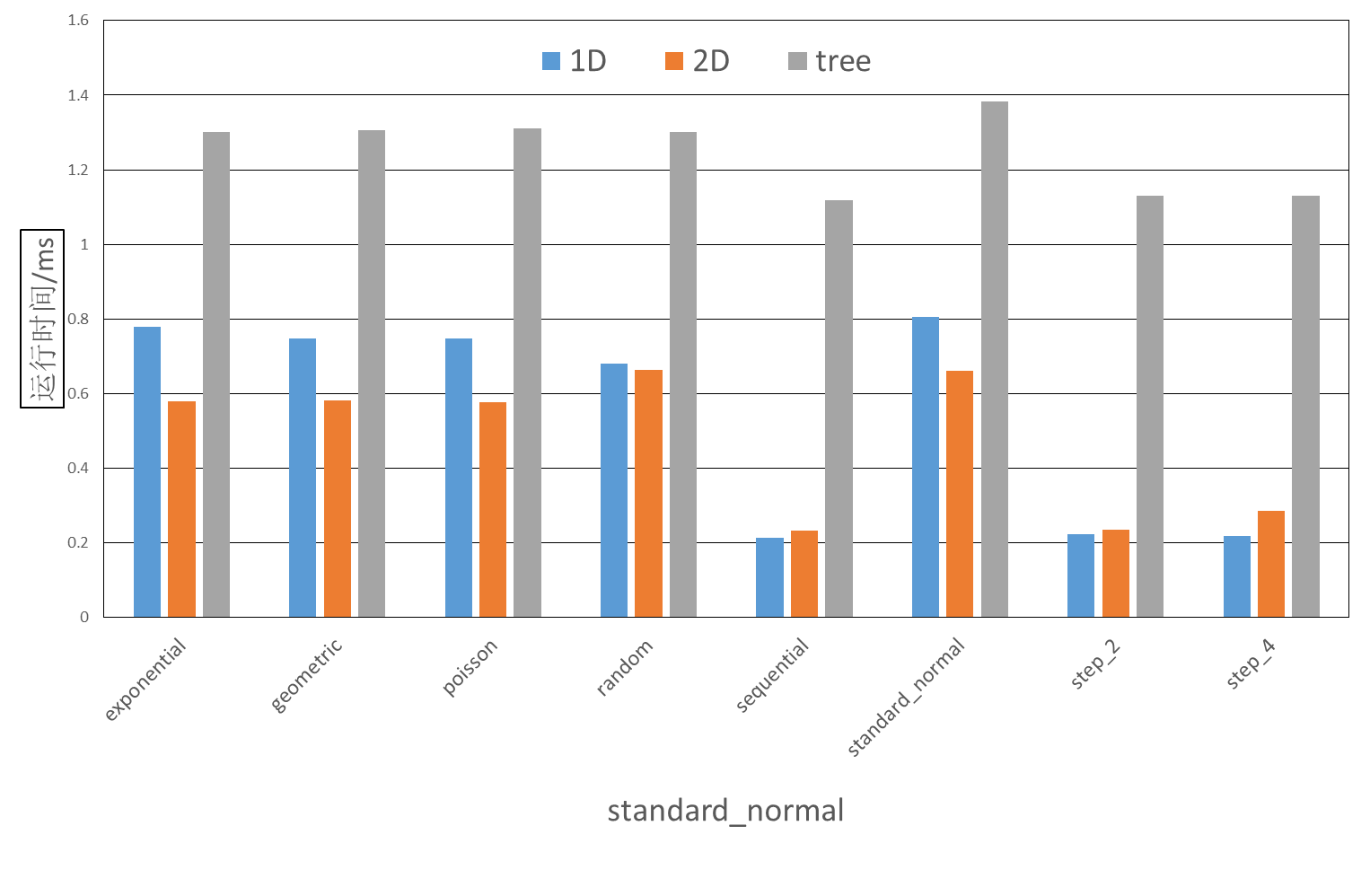
为了探究数据组织形式对数据访问性能的影响，我们选择合理的线程块大小和数据大小，内存存储位置选取全局内存，观察在不同数据内部分布和不同数据访问方式下，哪种数据访问方式能够获得最佳性能。实验结果如图\*\*，其中横坐标为数据内部分布，纵坐标代表运行时间。

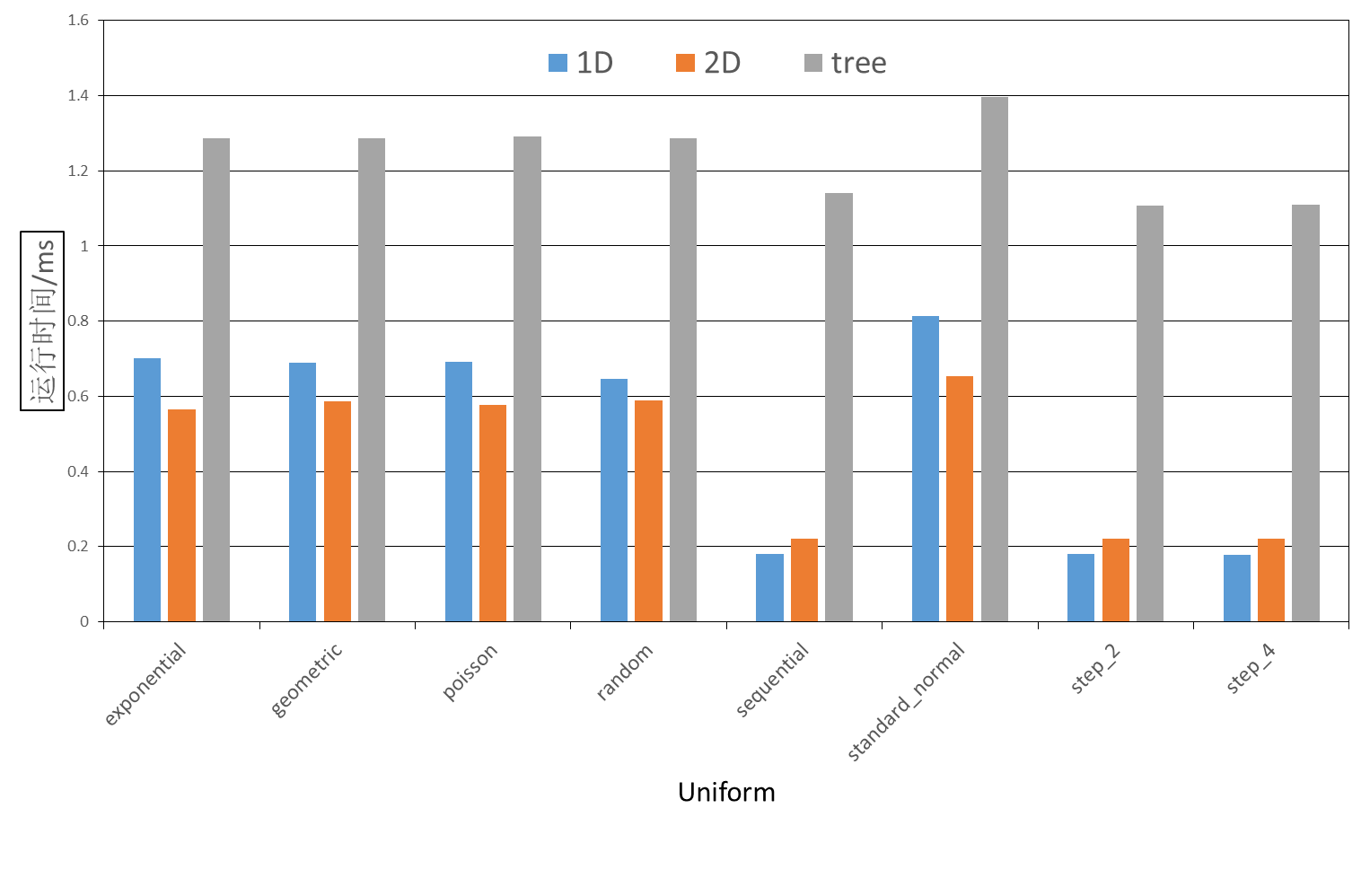








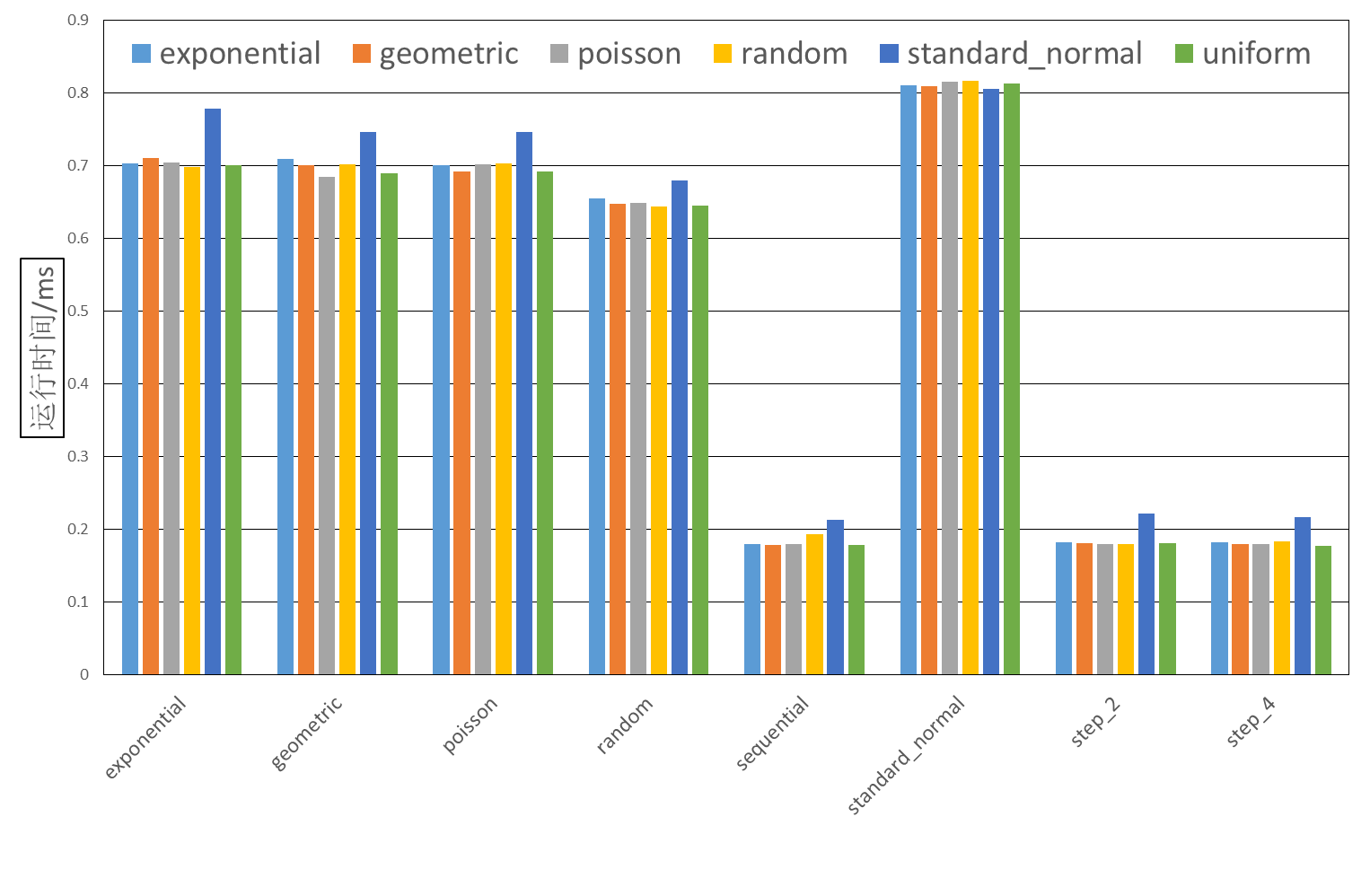


-

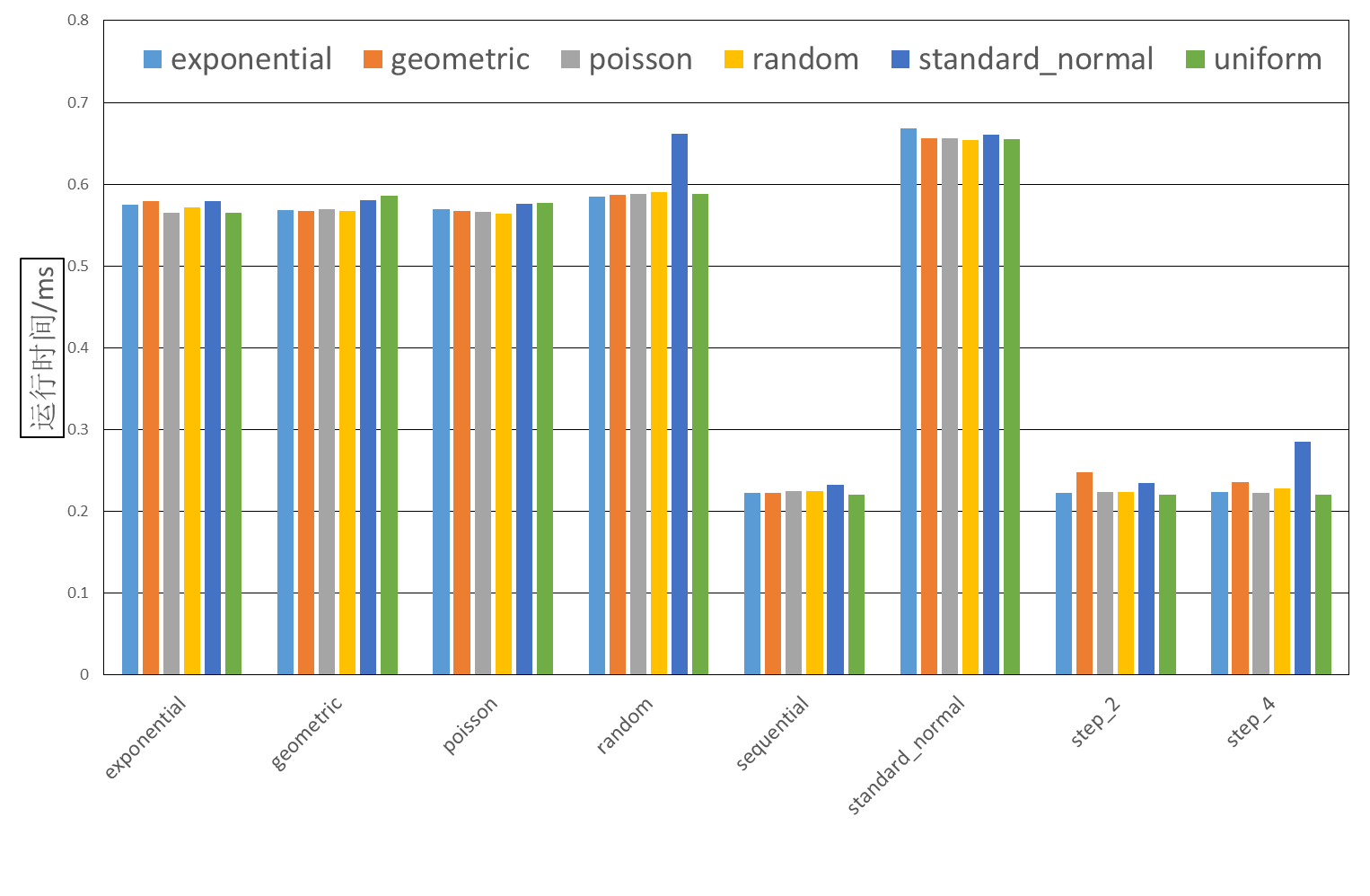
从图中可以看出，对于不同的数据组织形式，二维数组具有对齐访问优势，因此性能比一维数组好。而对于树结构，首先Node节点需要包含两个Node指针和一个数据，但数据节点大小较大，其次父节点与子节点之间往往“距离”较远，访问时对cache并不友好，所以性能最差。

### 3.2.4 Best data content

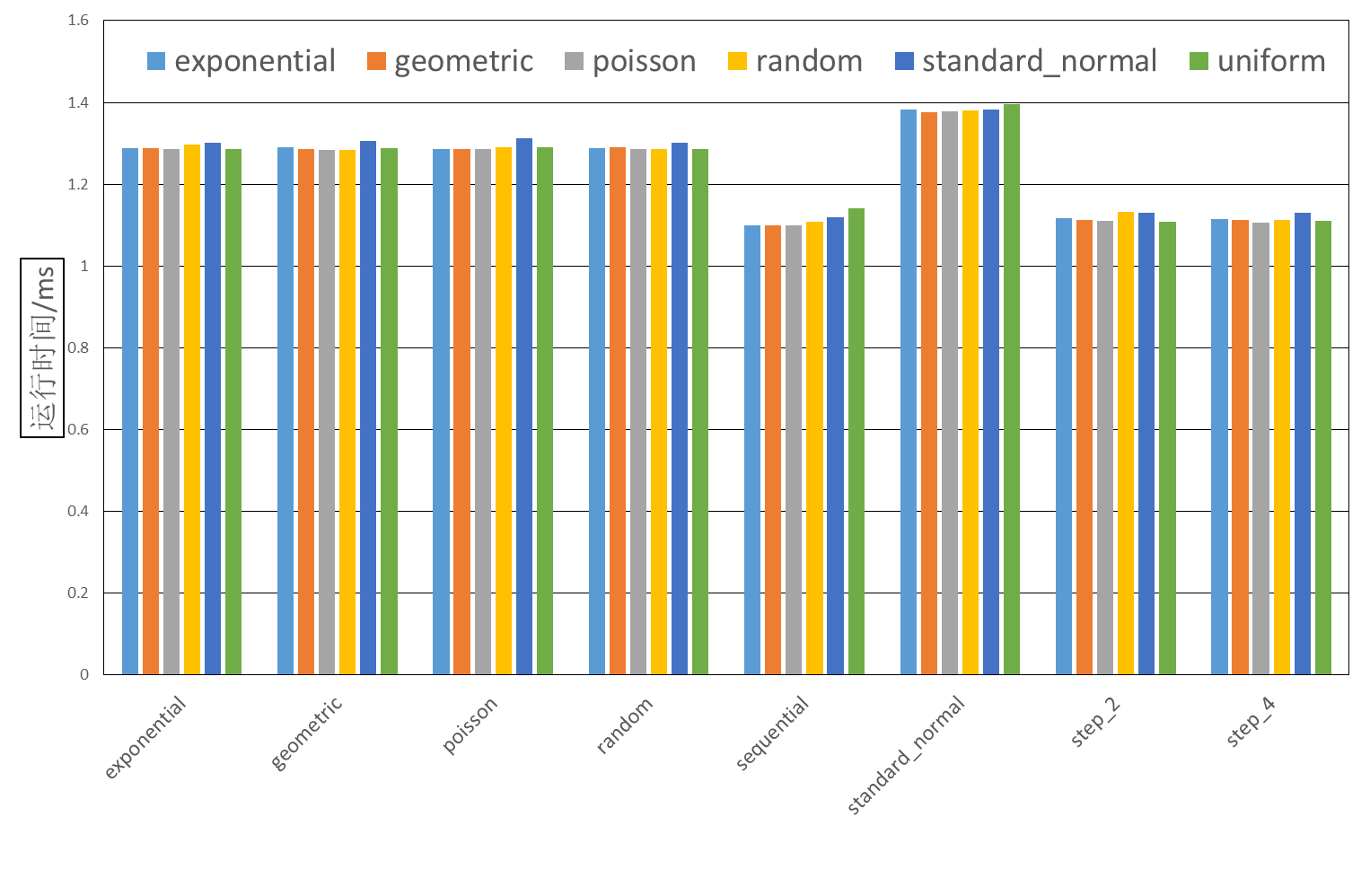
为了探究数据内部分布对数据访问性能的影响，我们选择合理的线程块大小和数据大小，内存存储位置选取全局内存，观察在不同数据访问方式和不同数据组织形式下，哪种数据访问方式能够获得最佳性能。实验结果如图\*\*，其中横坐标为数据访问方式，纵坐标代表运行时间。



一维数组



二维数组



树结构

从图中可以看出，对于不同的数据内容分布，所产生的数据对访问方式和数据组织对性能几乎没有影响。